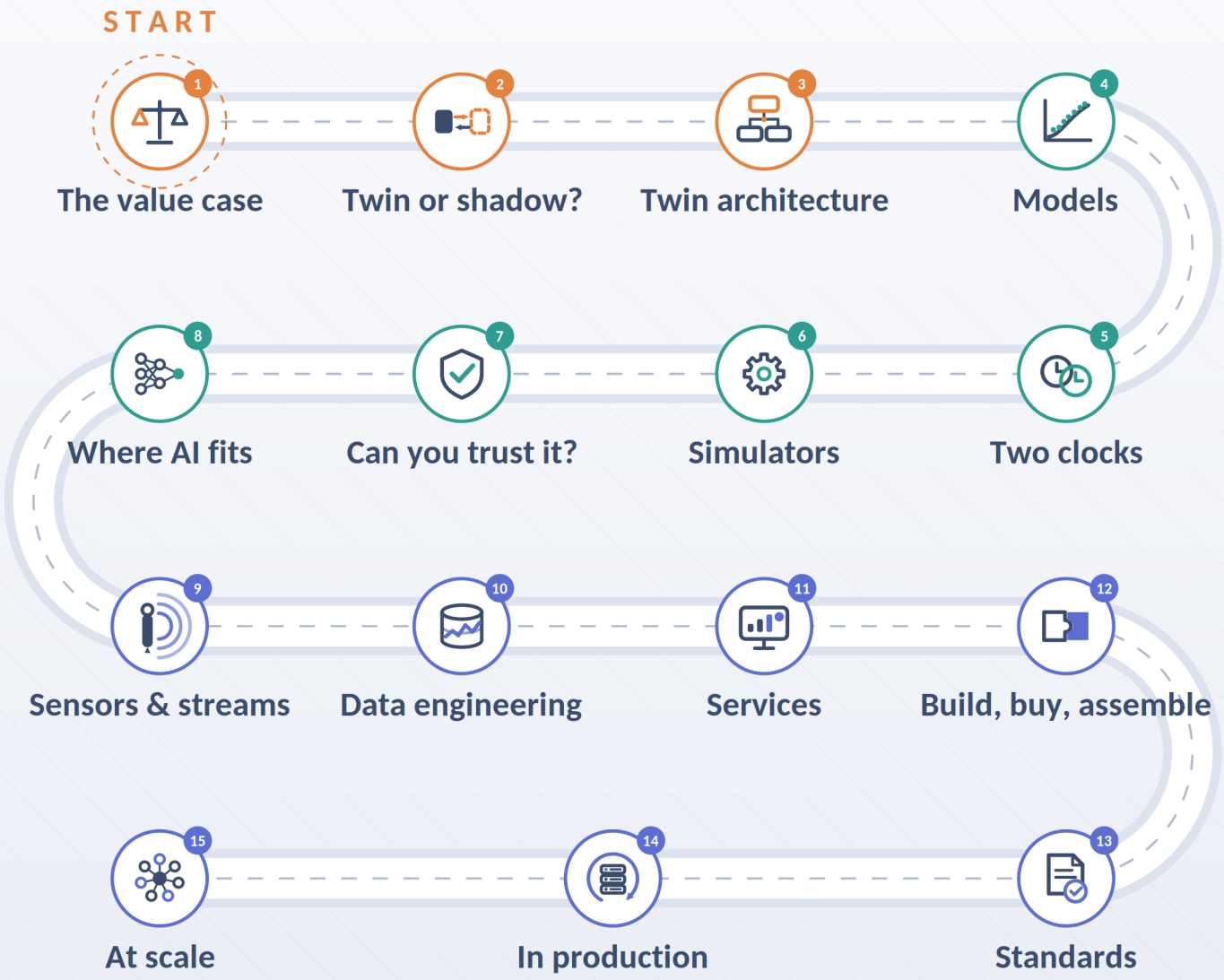
Digital Twins for Software Engineers

written by **Prasad Talasila**



Contents

I The Big Picture **1**

1 Why Anyone Pays for a Digital Twin **3**
1.1 Before You Start . . . 3
1.2 First Meeting . . . 4
1.3 Market Forecast . . . 5
1.4 Key Question . . . 6
1.5 Six Value Patterns . . . 11
1.6 Major Cost Contributors . . . 15
1.7 Application Areas . . . 16
1.8 Summary . . . 18
1.9 Exercises . . . 18
1.10 Further Reading . . . 20

2 Plant Demonstrator: A Case Study **23**
2.1 Before You Start . . . 23
2.2 Physical Twin . . . 24
2.3 Low-level Protocols . . . 29
2.4 Physical Interface . . . 33
2.5 Watering Strategies . . . 36
2.6 Plant Digital Twin . . . 41
2.7 Summary . . . 44
2.8 Exercises . . . 45
2.9 Further Reading . . . 47

3 Model, Simulation, Shadow and Twin **49**
3.1 Before You Start . . . 49
3.2 Model, Simulation and Simulator . . . 50
3.3 Digital Model to Digital Twin . . . 52
3.4 Non-Physical Systems . . . 58
3.5 Types of Digital Twins . . . 61
3.6 Misconceptions on Digital Twins . . . 63
3.7 Classification of the Plant Demonstrator . . . 65
3.8 Faded Examples . . . 69
3.9 Summary . . . 69
3.10 Exercises . . . 70
3.11 Further Reading . . . 73

4 Digital Twin Architecture **75**
4.1 Before You Start . . . 75
4.2 A Digital Twin Is a Software System . . . 76
4.3 Seven Building Blocks . . . 77
4.4 Four Families of Services . . . 82
4.5 Architectural and Design Principles . . . 84
4.6 The Layered Model: Physical Interface, Core, Digital Interface . . . 85
4.7 Lessons from Five Reference Models . . . 89
4.8 Digital Twin Configuration . . . 93
4.9 Architecture of the Plant Demonstrator . . . 96
4.10 Faded Examples . . . 101
4.11 Summary . . . 102
4.12 Exercises . . . 103
4.13 Further Reading . . . 105

II The Building Blocks **107**

5 Data for Digital Twins **109**
5.1 Before You Start . . . 109
5.2 The Twinning Loop, Read as Data . . . 110
5.3 A Vocabulary for Twin Data . . . 111
5.4 Sampling: From Physical Quantity to Number . . . 113
5.5 Sensors: What You Are Actually Given . . . 116
5.6 Time: Four Timestamps and Two Clocks . . . 117
5.7 Moving the Data: What a Twin Needs from a Transport . . . 120
5.8 Encodings: A Value in Flight . . . 126
5.9 Functions: The Computations Between Reading and State . . . 129
5.10 Processing: Streams, Windows and Placement . . . 133
5.11 Storing Twin Data . . . 135
5.12 Quality: The Field the Pipeline Carries . . . 140
5.13 How to Choose . . . 141
5.14 The Plant Demonstrator's Data Path . . . 143
5.15 Faded Examples . . . 145
5.16 Summary . . . 147
5.17 Exercises . . . 148
5.18 Further Reading . . . 151

6 Models **153**
6.1 Before You Start . . . 153
6.2 What Every Model Has . . . 154
6.3 The Running Example: An Offshore Wind Turbine . . . 158
6.4 Lumped-Parameter First-Principles Models . . . 160
6.5 Discrete-State and Discrete-Event Models . . . 164
6.6 High-Fidelity Field Models . . . 168
6.7 Reduced-Order and Surrogate Models . . . 170
6.8 Probabilistic Models and Bayesian Updating . . . 173
6.9 Keeping a Model Current . . . 176
6.10 Choosing a Model Type . . . 179
6.11 Modelling the Plant Demonstrator . . . 181
6.12 Faded Examples . . . 182

6.13 Summary . . . 185
6.14 Exercises . . . 186
6.15 Further Reading . . . 189

7 Simulation Techniques and Simulators 191
7.1 Before You Start . . . 191
7.2 What a Simulator Is, Architecturally . . . 192
7.3 Continuous-Time Simulation: Running the Rate Rules . . . 194
7.4 Discrete-Event Simulation: Running the Event Queue . . . 198
7.5 Hybrid Simulation: Where the Two Meet . . . 201
7.6 Stochastic Simulation: Many Runs Instead of One . . . 203
7.7 Field Solvers, and Why They Live Outside the Loop . . . 206
7.8 Making an Expensive Simulator Cheap . . . 208
7.9 Time: Simulated Clock, Wall Clock and the Twin's Clock . . . 211
7.10 Driving a Simulator from a Twin . . . 213
7.11 Choosing a Simulation Technique . . . 218
7.12 Simulating the Plant Demonstrator . . . 220
7.13 Faded Examples . . . 222
7.14 Summary . . . 225
7.15 Exercises . . . 226
7.16 Further Reading . . . 228

III The Ecosystem 231

IV Appendix 233

A The Mathematics You Need 235
A.1 Before You Start . . . 235
A.2 Quantities, Rates and Equations . . . 236
A.3 The Integration . . . 238
A.4 From an Equation to a Loop . . . 240
A.5 Many Quantities at Once . . . 243
A.6 Probability and Distributions . . . 249
A.7 State Machines with Probabilities: The Markov Chain . . . 253
A.8 What a Solver and a Simulator Do . . . 257
A.9 Notation Used in This Book . . . 259
A.10 Where Each Section Is Used . . . 262
A.11 Exercises . . . 264
A.12 Further Reading . . . 266

Bibliography 267

Part I

The Big Picture

Chapter 1

Why Anyone Pays for a Digital Twin

1.1 Before You Start

What you are assumed to know. You write software for a living. You are comfortable with HTTP APIs, JSON, a time-series or relational database, and reading Python you did not write. You have seen a system that talks to hardware, even if you did not build it.

What you are not assumed to know. Anything about control theory, numerical simulation, process chemistry, or the digital-twin literature. Where this chapter needs a term from one of those fields, it defines the term at first use and then keeps using it.

What this chapter covers. Why organisations spend money on digital twins; how that spending is justified before a line of code exists; what the published market numbers do and do not tell you; what six recurring shapes of value look like and where the human stands in each; where the cost sits; how four families of real deployments argue for their money; and how to build the value case for a small system, worked in full on a reactor vessel that is dosed with reagent on a timer.

What this chapter deliberately does not cover. What a digital twin *is*, technically. That sounds like the wrong order, and it is a deliberate choice. A later chapter draws the line between a twin, a shadow, a model and a simulation, and another gives the architecture. Both are easier to follow once you know what the thing is *for*. Until then, this chapter runs on one working sentence:

> **Digital twin (working definition, Part I only).** Software that keeps a model of one specific physical system in step with that system's actual state, and uses it to control that system and to answer questions about it that measurement alone cannot answer.

That sentence is deliberately loose. The literature does not agree on where a *digital twin* ends and a *digital shadow* begins, and a shared, transparent understanding of the two terms is still being argued for in manufacturing [1]. The tightening matters, because the boundary carries cost consequences, and a later chapter does it. For now the loose version is enough to talk about money.

Two more terms, pinned now and used unchanged for the rest of the book. The **physical twin (PT)** is the real system being twinned. In the worked example below that is the reactor vessel, its contents, the concentration sensor, the dosing pump, the tubing, and the controller that samples the sensors. The **digital twin (DT)** is the software defined above. From here on this chapter writes PT and DT rather than spelling either out, and so does the rest of the book. When the book says "the twin" with no qualifier, it means the digital one.

Learning objectives. By the end of this chapter you will be able to:

1. **Name the decision** a proposed twin is meant to improve, and turn a vague goal into one value metric with a unit, a denominator, a measurement method and a measured baseline.
2. **Classify** a proposed or existing system into one of six value patterns, state which rung of the ladder it sits on, and say what the next rung up would additionally require, including who acts on its output.
3. **Rank** the cost lines of a first-increment twin for a given system, and identify which line is most likely to be forgotten.
4. **Critique** a market forecast or a vendor claim by separating what was measured from what was projected, and reply to a request for a twin whose output nobody could act on.
5. **Construct** a complete value case – decision, metric, baseline, pattern, fidelity requirement, cost and payback – for a small cyber-physical system, and state the one test that could falsify it before any code is written.

Every section that follows serves at least one of those five, and the exercises at the end say which.

1.2 First Meeting

Here is the situation this chapter is written for.

You are a software engineer. Someone senior has decided that your organisation needs a DT. There is a kick-off meeting. Around the table are a modelling expert who talks about solvers and boundary conditions, a business sponsor who talks about uptime and margin, and a customer who talks about the thing they own and worry about. You are there because someone has to build it.

The failure mode of that meeting is not technical. It is that everyone agrees enthusiastically while meaning four different things, and nobody says out loud which decision the twin is supposed to improve. Six months later the project has a beautiful three-dimensional model, a live data feed, a dashboard, and no answer to the question “what did we do differently because of it?”

This is not a hypothetical failure. Writing about supply-chain twins, one practitioner warns that new technology reads as an expensive gamble unless whoever is buying it keeps their hopes for the payoff, and for how soon it arrives, realistic [2]. A set of pilot reports from European manufacturing points the same way from the other side: the company that describes its pilot as a success says it had worked out, before starting, why it wanted the twin and what it would be used for [3]. A brief from an industry consortium adds the part that is hardest to hear in a kick-off: the process around the twin has to change for the benefits to arrive, because a twin bolted onto an unchanged process does not deliver them [4].

So the first skill this book teaches is not modelling and not architecture. It is the ability to sit in that meeting and ask, without embarrassment: **which decision gets better, who makes it, and how will we know?**

1.2.1 Where the idea came from, in one paragraph

You will hear the origin story in meetings, so it is worth having straight. The concept traces to a 2002 presentation to industry, given at the University of Michigan, in which Michael Grieves proposed a real space, a virtual space, and a data link between them for product lifecycle management; the early courses called it the Mirrored Spaces Model [5]. The name “digital twin” attached later, and the 2010 technology roadmap of the National Aeronautics and Space Administration (NASA) gave the first detailed definition, in the aerospace context [6]. The folklore version – that NASA’s Apollo programme built two identical vehicles, flew one and kept the other on the

ground to rehearse fixes – is real enough as history and is routinely cited as the ancestor of the idea [7]. The defence-side literature makes the same point without the twin vocabulary: Apollo leaned on simulation throughout, to keep the risk down for the astronauts and for the people running mission control [8].

Note what the Apollo story is really about. It is not about fidelity for its own sake. It is about moving a decision – "will this fix work?" – from a place where being wrong is fatal to a place where being wrong is cheap. Nearly every DT value argument in this chapter is a variant of that one move.

1.3 Market Forecast

You will be shown a market forecast. Here is how to read one without either swallowing it or sneering at it.

The published figures are large and mutually inconsistent, and the four that reach this corpus through academic surveys are set side by side in Table 1.1 (figures in US dollars (USD)). Each row is a paper quoting an analyst house; none of the four papers measured a market.

Table 1.1: Four digital-twin market forecasts, as relayed by papers in this corpus. Every figure is an analyst projection quoted second-hand.

Quoted in	Analyst house	From	To	Growth quoted
A 2020 state-of-the-art survey [9]	Grand View Research	USD 2.26 billion (2017)	USD 27.06 billion (2025)	"approximately tenfold"
A 2022 enabling-technologies survey [10]	not named in the survey	USD 3.2 billion (2020)	USD 48.27 billion (2026)	58% per year
A 2022 modelling review [11]	Markets and Markets	USD 3.1 billion (2020)	USD 48.2 billion (2026)	not quoted
A 2025 twinning-systems paper [12]	Fortune Business Insights	USD 8.6 billion (2022)	USD 137.7 billion (2030)	42.6% per year

Read the table down the "To" column. Two forecasts for the same year, 2026, agree with each other to within a rounding error, which is less reassuring than it looks: they are the same analyst estimate reaching two papers by different routes. Set the 2025 and 2030 figures beside them and the implied size of the same market in any one year moves by more than a factor of two depending on who counted. Four observations follow, in order of usefulness to you:

1. **These are forecasts, not measurements.** Cite them as evidence of expectation, not of size. The same survey that relays the 58% growth figure also notes that Gartner named DTs a top-ten strategic technology trend for 2019 [9], which is evidence of attention, not of revenue.
2. **The disagreement is the information.** When "digital twin" can mean a dashboard or a coupled multi-physics simulation, any total is a total over an ill-defined set. That is why the definitional chapter exists.

3. **Adoption lags the forecasts, and the adoption numbers also disagree.** The 2022 survey relays an engineering-institution report putting industry-agnostic adoption at about 5% of enterprises [10], while the 2025 paper relays surveys in which only about 5% of companies say a twin plays *no* role in their digital-transformation plans [12]. Both can be true. "We have a strategy" and "we have a twin in production" are different claims, and holding both in mind is the correct posture.
4. **Sector reports beat global totals.** Oil and gas [13], health [14], smart cities [15] and net-zero manufacturing [16] each have their own literature with concrete application scenarios. A sector report tells you what people in *your* customer's industry are doing. A global total tells you what a research firm sold a report about.

Two further pieces of context are worth carrying into the meeting. Several years of digital-transformation progress are credited to the COVID-19 period alone [10], so part of the growth in these curves is a step change in attention rather than a smooth technology trend. And vendor material is part of the evidence base: an accessible overview of the field draws on what large vendors report doing with twins [17]. That is not disqualifying, but it means the loudest numbers come from parties with an interest in them.

What to say in the meeting. Not "the market will be USD 137 billion". Say: "forecasts are large and inconsistent, adoption is early, and our justification has to come from our own baseline, not from a growth curve." That sentence is both more defensible and more useful.

1.4 Key Question

Start from a claim that will look reductive and then earn its keep:

> **A DT is bought to improve a decision. If you cannot name the decision, there is no value case – only a budget.**

A decision has four parts you can write down in a meeting:

1. **The decision itself.** "How much reagent does vessel 7 get this shift?" "Do we take turbine 12 offline this week or next month?" "Do we accept this batch?"
2. **The decider, and the cadence.** A human once a quarter, a human every morning, or a control loop every 200 milliseconds. These are three different systems, and the cadence drives most of your architecture.
3. **What is wrong with the decision today.** It is made too late, or on incomplete information, or by a person who has to be physically present, or it is not made at all because nobody noticed there was a decision to make.
4. **What it costs to be wrong.** In currency (this chapter's worked examples are costed in euros, EUR), in downtime, in scrapped product, in a scrapped batch, in a life.

Point 4 is the one that gets skipped, and it is the one that determines whether the project is worth doing. A twin that improves a decision whose error cost is EUR 200 a year is a hobby.

1.4.1 The value metric

From those four parts you extract exactly one **value metric**: a measurable quantity, with a stated baseline and a stated way of measuring it, that the twin is supposed to move.

Insisting on *one* metric is the core of what practitioners call the **lean digital twin**: pick the single quantity the business most wants moved, attach a money value to moving it, build only what is needed to show the movement within weeks rather than years, and let that early return

pay for the next, more capable increment [18]. That is a familiar shape to any engineer who has shipped a minimum viable product, and the analogy is the authors' own. The business viewpoint of an industrial reference architecture asks for the same thing in more formal clothes: a return, with costs and benefits tracked across the whole solution lifecycle – development, operation, evolution and maintenance, not the build alone [19].

Bad value metrics are abstract nouns: *efficiency*, *visibility*, *optimisation*, *digital transformation*. Good value metrics have units and a denominator, and Table 1.2 shows the difference on four requests you will actually hear.

Table 1.2: Requests as they arrive in a meeting, and the value metric each one hides.

A request as it arrives	The value metric behind it
"Improve maintenance"	Unplanned downtime hours per turbine per year
"Better dosing"	Batches scrapped because of undetected dosing faults per year
"More visibility"	Hours between a fault occurring and an engineer being told
"Optimise the line"	Scrapped units per 10,000 produced

The right-hand column has a property the left does not: someone can go and measure it *today*, before you build anything, and get the **baseline**. A value claim with no measured baseline is subtraction with one operand missing.

Three quantities carry the rest of this chapter, so pin them now. Write the value metric as M, its measured baseline as M_0, the value the twin is expected to reach as M_1, and the cost of one unit of the metric as c. The **annual benefit** is then

$$B = (M_0 - M_1)\, c \tag{1.1}$$

and the **payback period**, in months, is the build cost plus one year of running cost divided by that benefit:

$$P = 12\, \frac{C_{\text{build}} + C_{\text{run}}}{B} \tag{1.2}$$

Both are arithmetic a sponsor can check on a napkin, which is the point. Equation 1.1 is where most of the argument happens, because M_1 is a forecast and c is a judgement, and Equation 1.2 is where the argument is tested, because C_{build} and C_{run} are things you will be held to.

1.4.2 Decision latency: what twins usually actually sell

Look across the deployments in the Application Areas section and one quantity keeps reappearing: **decision latency**, the time between the PT entering some state and someone, or something, acting on it. Occasionally a twin opens up a decision nobody could have made before, because it offers a service that did not exist otherwise. Far more often it makes an already-possible decision happen *hours or weeks earlier*, or *without a human standing next to the equipment*.

Decision latency splits into two gaps, and a DT does not touch both of them equally. Figure 1.1 separates them.

Write the two as G_{d}, the **detection gap** from the state being entered to somebody noticing, and G_{r}, the **response gap** from noticing to acting. Then

$$L = G_{\text{d}} + G_{\text{r}} \tag{1.3}$$

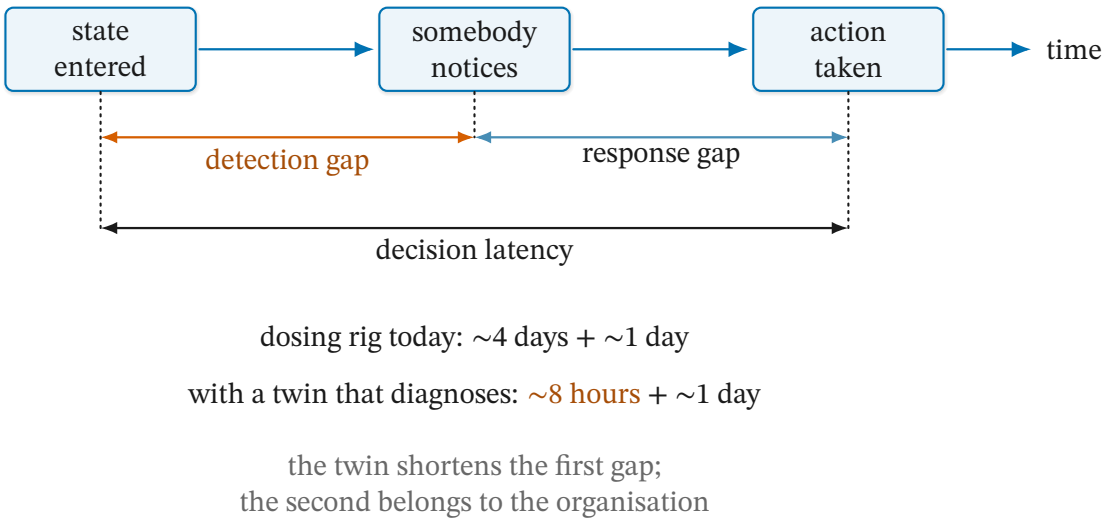


Figure 1.1: Decision latency, split into the detection gap a twin shortens and the response gap it does not.

Equation 1.3 looks too small to write down, and its value is in what it separates. The detection gap is an engineering problem and yours to solve. The response gap is an organisational one – how fast the lab, the crew or the maintenance schedule can act once told – and no amount of model fidelity shortens it. This is the concrete form of the consortium brief's warning that the process has to change for the benefit to arrive [4]: a twin that cuts G_{d} from four days to eight hours has done nothing for a plant whose G_{r} is a fixed weekly maintenance round. It is why the *predict* pattern in the next section carries a warning, and why "nobody can act on the output" appears below among the reasons to stop.

That split is worth drawing in the kick-off meeting, because it reframes the project from "build a virtual replica" – unbounded, expensive, un-finishable – to "cut the time from event to action from four days to four hours" – bounded, estimable, testable. The lean-twin literature's insistence on a benefit that can be verified quickly [18] is the same reframing from the finance side.

1.4.3 Three examples, with the support fading

The method above is small enough to run in an afternoon. Here it is run three times: once in full on a small dosing rig, once on a larger system with one step left to you, and once posed as a problem for the exercises.

Example 1, worked in full: a reagent dosing rig

Picture a pilot plant with twelve reactor vessels. Each vessel has an inline concentration sensor and a small dosing pump fed from a shared reagent tank, all wired to one controller that stores every reading in a time-series database and exposes a REST (Representational State Transfer) API. Nothing about it is exotic; a brewery, a water-treatment works or a fish farm has the same shape. For the value case, one fact about it matters, and Figure 1.2 shows it.

Step 1 – the decision. Dosing is on a fixed shift schedule: 50 ml at 06:00, 14:00 and 22:00, whatever the sensor says. The decision "how much reagent, and when" is made once, by whoever set the timer, and then not made again. The sensor readings are stored and never consulted by anything that acts. So the decision that exists today is not "how much reagent" but "has dosing

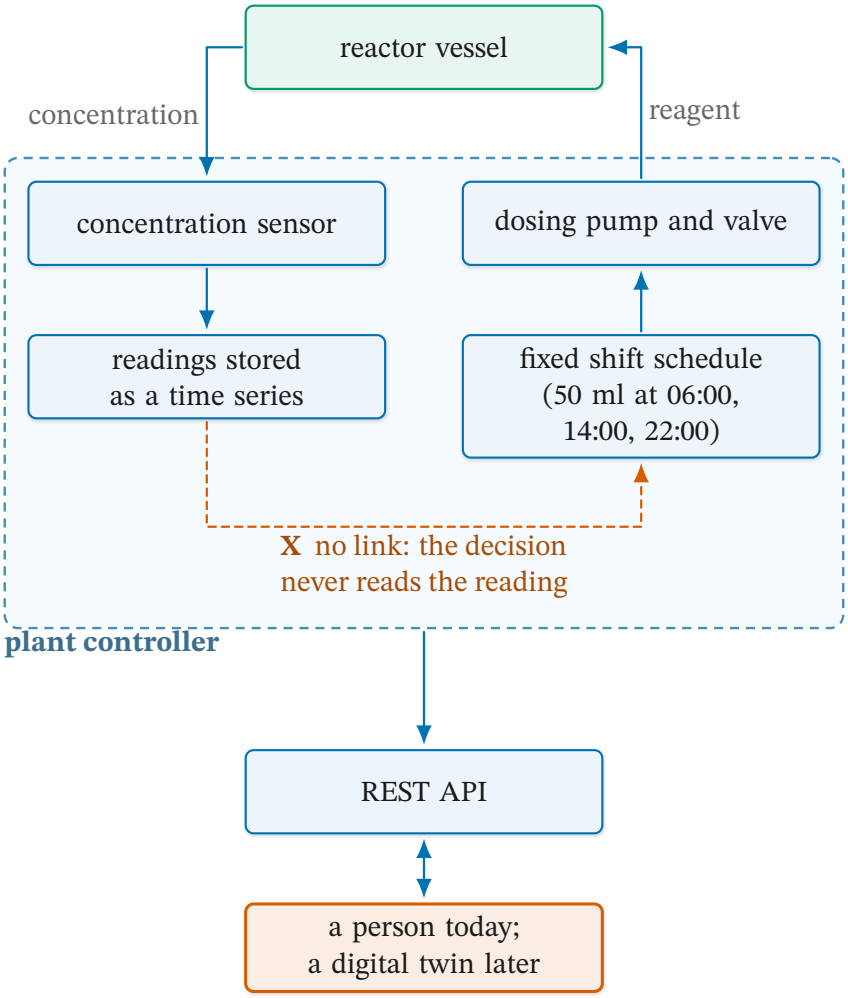


Figure 1.2: The dosing rig as it exists today. Sensing works and actuation works; the dosing decision never reads the measurement. The missing link, marked X, is what this value case is about.

stopped working": a blocked line, an empty tank, a valve that no longer opens. Today a person makes that decision when the end-of-batch assay comes back out of specification.

Step 2 – the decider and the cadence. A shift technician, on their rounds, roughly daily. The reasoning for writing this down: it fixes G_r. Once told, the technician can act within about a day, and no software changes that.

Step 3 – what is wrong with it today. The decision is made late. A dosing fault shows in the assay after about four days, so G_d is about four days and L is about five. The rig's own log confirms it: the stored concentration series shows flat traces for days before each recorded intervention.

Step 4 – what being wrong costs. This is where the obvious metric fails and the real one appears. The obvious metric is reagent. Twelve vessels at three doses of 50 ml a day is $12 \times 0.05 \times 3 \times 365 = 657$ litres a year, which at the reagent's bulk price is a few hundred euros. Nobody funds a project to save that. The real cost is that each vessel runs a six-day batch, and a batch that went undosed for more than about two days is scrapped and restarted. Last year's log records ten dosing faults and, counting the batches scrapped because of each, 37 batches lost. The plant costs a batch – materials plus vessel time – at about EUR 800. So the metric is *batches scrapped because of undetected dosing faults per year*, with $M_0 = 37$ and $c = 800$, and the baseline cost of the problem is $M_0\, c = 37 \times 800 = 29{,}600$ euros a year. The reasoning for choosing this metric over reagent: it is the quantity whose reduction the sponsor would recognise as the point.

Step 5 – the target. A twin that compares each dose's expected effect on the concentration with the recorded effect would see a fault at the first missed dose, within one dosing cycle of about eight hours. With G_{d} at eight hours and G_{r} at a day, the batch survives. Not every fault is visible this way – a sensor that fails silently is the obvious hole – so set the target conservatively at $M_1 = 4$. Then, by Equation 1.1, $B = (37 - 4) \times 800 = 26{,}400$ euros a year. The reasoning for the conservative target: a value case that survives its own pessimism is one you can defend in the meeting.

Step 6 – the pattern and the fidelity it needs. The twin has to notice that a dose did *not* do what a dose does. It does not need to forecast the concentration a week ahead, and it does not need to decide the dose. In the vocabulary of the next section that is *diagnose*, not *predict* and not *control*, and the model it requires has to get the direction and rough size of a dose's effect right – nothing more. The reasoning: fidelity is being derived from the metric, not chosen for its own sake.

Step 7 – cost and payback. Three engineer-weeks to build the comparison, alerting and a small calibration harness, plus half a week of a modeller's time for the dose model, at the plant's internal rate: $C_{\mathrm{build}} = 14{,}000$. Running cost a year: hosting at a few hundred euros, plus half a day a month of upkeep as vessels, tubing and sensors are swapped: $C_{\mathrm{run}} = 5{,}400$. By Equation 1.2, $P = 12 \times \frac{14{,}000+5{,}400}{26{,}400} = 8.8$ months.

Step 8 – the test that could kill it. Before writing any code, spend one afternoon querying the stored concentration history for the ten logged faults. If the traces do *not* go flat at the first missed dose – if the sensor noise swamps a 50 ml step, or the faults were of a kind the sensor cannot see – then M_1 is not 4, the eight-hour detection gap is fiction, and the case collapses. That afternoon costs nothing and can falsify the whole argument. Do it first.

Example 2, one step left to you: a pump fleet

A water utility runs forty similar pumps across a city. Unplanned outages are logged: last year they totalled 1,200 pump-hours, so $M_0 = 1{,}200$ with the metric *unplanned outage hours per year across the fleet*. Each outage hour costs the utility about EUR 250 in standby crew and overflow penalties, so $c = 250$. The decision is "which pump do we visit today", made every morning by a maintenance planner, and today it is made from whichever pump complained loudest overnight.

A twin that flags a pump whose vibration and power draw drift from what its model expects – *diagnose* again – would let the planner visit before the failure. Engineering judgement, checked against the log, says about 40% of outage hours were preceded by a detectable drift, so set $M_1 = 720$. Then $B = (1{,}200 - 720) \times 250 = 120{,}000$ euros a year. Integration with forty pumps of three vintages, a vibration model per vintage and a year of running cost come to $C_{\mathrm{build}} = 90{,}000$ and $C_{\mathrm{run}} = 30{,}000$.

Your step: compute P from Equation 1.2, and then answer the question the payback does not: what is G_{r} here, and what would make the whole case fail even if the payback looks acceptable? A check appears in the exercise solutions.

Example 3, posed as a problem: a municipal park

A parks department irrigates a large public park from a borehole on a fixed schedule. Water is metered and paid for. Turf is replaced when it dies. Complaints arrive by e-mail. The head of department has asked for "a digital twin of the park". Building this value case in full – decision, metric, baseline, target, pattern, fidelity, cost, payback, and the test that could kill it – is the hardest exercise at the end of this chapter, and it is deliberately not worked here.

1.4.4 When the right answer is no

The method has a feature its enthusiasts underplay: run honestly, it often returns "don't". Eight conditions, any one of which should stop the project or shrink it to a rung that pays:

1. **No decision can be named.** The request is for visibility, insight or transformation, and nobody can say what would be done differently.
2. **The error cost is trivial.** The metric is real but c is small enough that B never covers C_{run}.
3. **Nobody can act on the output.** G_{r} is fixed by a schedule, a contract or a staffing level, and shortening G_{d} changes nothing.
4. **The baseline cannot be measured.** No log, no history, no way to get M_0 without first building the thing that was supposed to justify itself.
5. **The falsifying test already failed.** The stored history does not show the signal the twin would need.
6. **A cheaper rung already delivers the benefit.** A threshold alarm on the existing sensor gets most of B at a fraction of C_{build}.
7. **The model cannot be kept valid.** The PT changes faster than anyone will re-calibrate; the next section on cost says why this line is the one most often forgotten.
8. **The organisation has not agreed to change its process.** The brief on infrastructure twins is blunt that the twin on its own buys you nothing; somebody still has to be willing to work differently [4], and the pilot reports single out a clearly stated purpose as what separated the pilots that worked [3].

None of these is a reason to be against twins. They are the conditions under which the *lean* literature's own logic – fund the increment that verifies quickly [18] – says there is no such increment yet.

1.5 Six Value Patterns

Across the deployment reports in this corpus, the *shapes* of value repeat. This book uses six names for them, and will use them consistently from here to the last chapter. They form a rough ladder: each rung typically needs everything below it, and each rung costs more than the one below. Figure 1.3 is the whole section on one page; the subsections that follow are its rungs, in order.

The ladder is a cost ordering, not a quality ordering. Rung 2 delivered is worth more than rung 4 proposed.

1.5.1 Pattern 1 – Monitor

"What is happening right now, and what happened last Tuesday?"

The twin ingests live measurements, stores them with context, and presents current and historical state. It may fill in quantities that are not directly measured: a **virtual sensor** is a model-derived estimate of a quantity with no physical sensor of its own – a weld's stress, say, that you cannot instrument directly – computed from the model's internal state [20]. A review of the field's evolution puts monitoring first historically too: twins began as monitoring systems and were later extended towards control, optimisation and autonomy [21].

- **Buys you:** lower decision latency, remote access, an audit trail.
- **Costs you:** connectivity, data engineering, and the unglamorous work of keeping timestamps and units correct.
- **Honest warning:** many systems sold as DTs stop here.

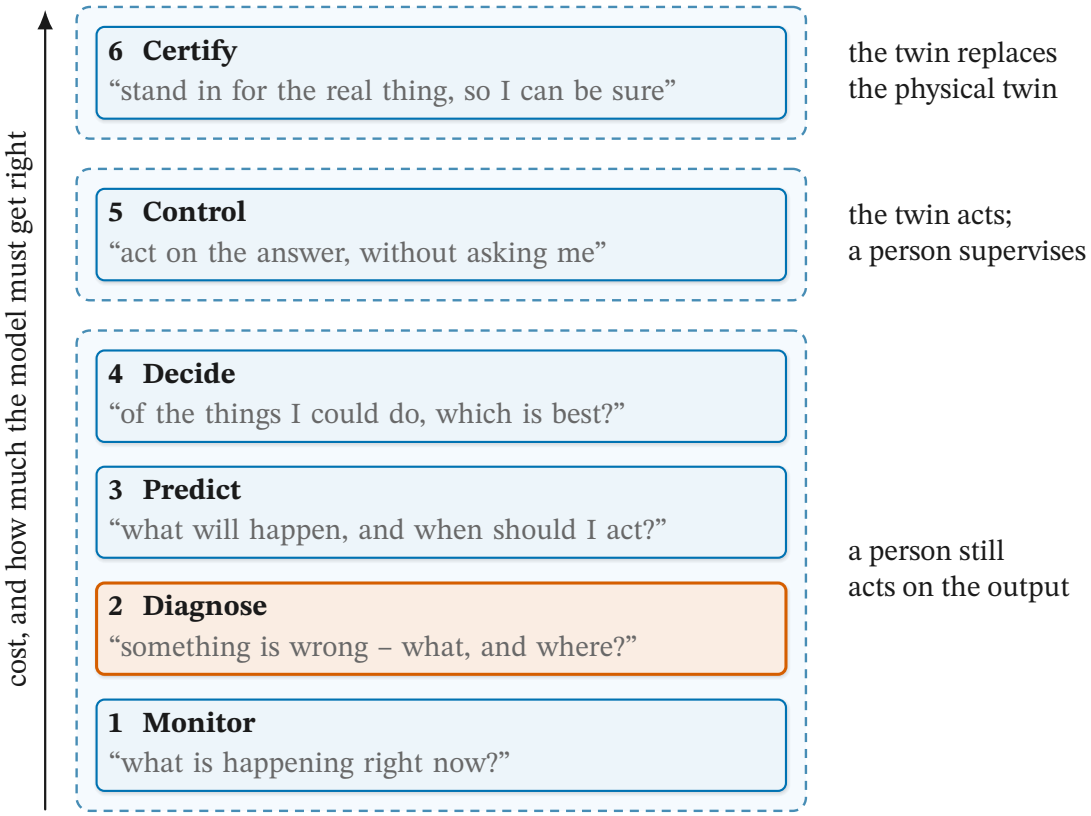


Figure 1.3: The six value patterns as a ladder. It is a vocabulary, not a maturity model: stop at the rung that pays for itself. The dashed groups mark who acts on the twin's output.

1.5.2 Pattern 2 – Diagnose

"Something is wrong. What, and where?"

The twin compares what the model says *should* be happening with what the sensors say *is* happening. The difference is the **residual**:

$$r(t) = m_{\text{obs}}(t) - m_{\text{model}}(t) \tag{1.4}$$

where $m_{\text{obs}}(t)$ is the measured value and $m_{\text{model}}(t)$ the model's expectation at the same instant. A large, structured residual localises a fault; the simplest diagnostic rule flags one when $|r(t)| > \theta$ for longer than the model's own noise would explain, with the threshold θ set from the stored history. Choosing θ well is later work; what matters here is that Equation 1.4 is the entire mechanism, and it needs a model that gets the *direction and rough size* of a change right, not one accurate to three decimal places.

Figure 1.4 shows the rule at work on the dosing rig from the previous section. After each dose the model expects a step in concentration followed by a slow decline; the sensor reports a flat line. Three doses, three missing steps, three large residuals – the reagent is not reaching the vessel, and the fault is visible one dosing cycle after it starts rather than when the assay comes back.

Floating offshore wind's diagnostic twins run the same pattern: the twin watches asset condition for anomalies on equipment that costs a lot, and takes a while, to reach in person [22], and behind fault-diagnosis twins for rotating equipment more generally [23]. It is also the level a systems-of-systems maturity ladder for asset twins calls *adaptive*: the twin changes as the physical reality changes, within an acceptable uncertainty, and is enhanced with diagnostic capability

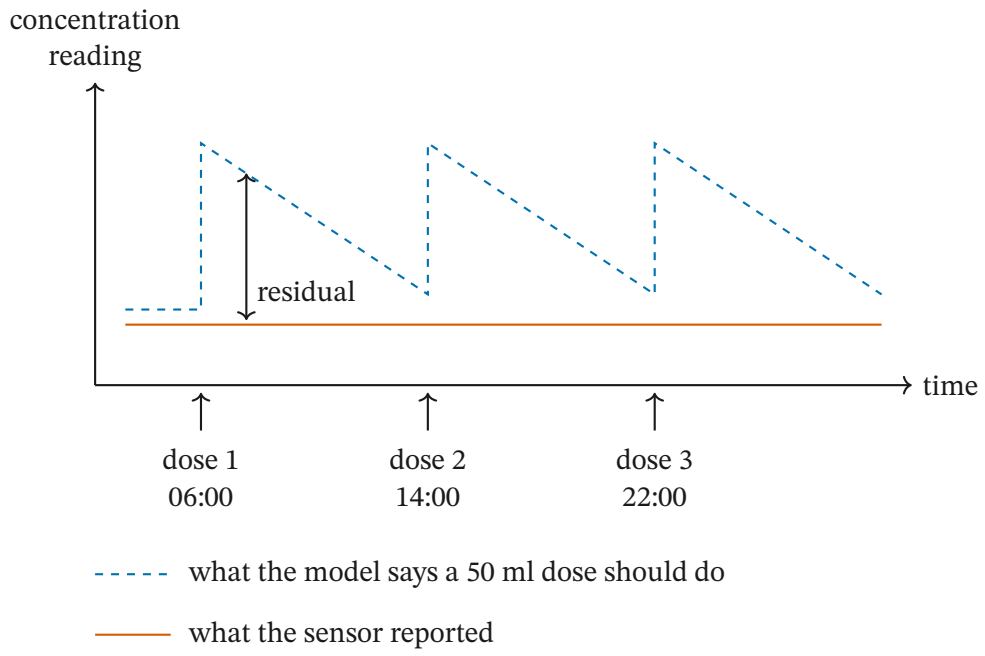


Figure 1.4: Why *diagnose* pays on the dosing rig. Each gap between the two traces after a dose is a residual, and the fault is visible within about eight hours rather than four days.

[24].

- **Buys you:** faults caught in hours instead of days; fewer truck rolls to find out nothing is wrong.
- **Costs you:** a model good enough that a residual means something, plus the calibration work of a later chapter.
- **Note for the meeting:** diagnosis is the cheapest rung with real engineering value, and it is very often the right first increment.

1.5.3 Pattern 3 – Predict

"What will happen, and when should I act?"

The twin runs the model forward from the current estimated state to forecast future condition: remaining useful life, time to failure, time until a vessel's concentration drifts out of specification. This is by far the most-reported family in the literature: predictive maintenance built on DTs has enough of its own literature to have been reviewed more than once [25, 26], and the maturity ladder above names this level *intelligent* – predictive and prescriptive capability layered on the adaptive twin [24].

- **Buys you:** maintenance scheduled by condition instead of by calendar; interventions before the failure rather than after.
- **Costs you:** everything in *diagnose*, plus a model that stays accurate over a forecast horizon, plus a serious answer to "how wrong might this be?" Uncertainty quantification stops being optional here [27].
- **Honest warning:** a prediction nobody acts on has zero value. If the maintenance crew's schedule is fixed six months ahead, a two-week-ahead prediction changes nothing. Check G_{r} before you sell a shorter G_{d}.

1.5.4 Pattern 4 – Decide

"Of these five things I could do, which is best?"

The twin runs many alternative futures and compares them. The literature calls this **what-if simulation**: a data- and computation-intensive approach that evaluates how the PT would behave under different scenarios and hypotheses, so that when something unwanted shows up an operator – or the twin itself – can simulate the candidate interventions, compare the outcomes, and choose between them on better information [28]. The output is a recommendation; a person still chooses.

A **fleet** is a set of similar PTs, each with its own DT, managed together, and the fleet level is where large operators claim their money back. A reported analysis of a 125-turbine fleet costs one component's early warning against the whole fleet rather than the single machine, and that reframing is what turns cost control from a per-unit exercise into a plant-wide one – weighed against revenue, across the fleet [29].

- **Buys you:** better decisions, and decisions a human could not have enumerated by hand.
- **Costs you:** simulation you can run *fast* and *many times*. This is the first rung where compute cost becomes a design constraint rather than a line item: a what-if simulation is only useful if it runs faster than real time, and a documented incubator twin answers a power-outage plan by running three whole simulations, one per candidate controller setting, while the outage is still under way [28].

1.5.5 Pattern 5 – Control

"Act on the answer, without asking me."

The twin closes the loop: it changes the PT's behaviour directly, by reconfiguring it or by issuing set-points, and a person supervises rather than decides. The chapter on advanced services treats effecting change in the PT through reconfiguration as the natural end of what-if analysis [30], and the maturity ladder calls this level *autonomous*: the twin operates in automated closed-loop mode on the asset's own data, with autonomous functions built gradually on top of the automated ones [24].

The dosing rig's own closed loop – reading the concentration and setting the dose – is the obvious thing to build, and it is worth noticing that the value case in the previous section did *not* ask for it. Closing the loop would save a little reagent and would not shorten G_{d} for the faults that scrap batches. That is the ladder doing its job.

- **Buys you:** a decision that happens at the cadence of the PT rather than of a person – the 200-millisecond loop from the Key Question section.
- **Costs you:** everything below, plus the safety argument. The same systems-of-systems chapter that names the autonomous level also flags it as the contested one, and says it currently only fits cyber-physical systems that already have some autonomy built in [24]. If the twin is wrong, the PT does the wrong thing, unattended.

1.5.6 Pattern 6 – Certify

"Can I be confident before I touch the real thing?"

The twin substitutes for the PT when using the PT is expensive, slow, dangerous, or not yet built. The sub-case that matters most to you as a software engineer is **testing the software that controls the PT**. A reproducible exemplar built around a small robot vehicle uses a twin prototype precisely so that developers can swap a hardware-in-the-loop rig for a software-in-the-loop one and keep working on the same embedded software in parallel, without the special

hardware environment that embedded testing usually demands [31]. Teams interviewed about continuous integration on cyber-physical systems report doing exactly this in their production pipelines – in one case relying on third-party simulators supplied by the customer because the hardware-in-the-loop rig lives in a clean room the pipeline cannot reach [32].

The second sub-case is **evidence**. Defence-side digital engineering is defined around an authoritative source of system data and models across the lifecycle, so that analysis done on the models counts as evidence about the system [8]. In a regulated domain that is what "certify" means: the twin's output is admissible.

- **Buys you:** a test environment, a rehearsal space, and – in regulated domains – evidence.
- **Costs you:** credibility. If decisions rest on the twin's output, you must be able to say why it should be believed. That is what most projects underestimate.

1.6 Major Cost Contributors

Twins are not free, but the return can be large – that is the industrial sales pitch in one line, and it is a fair summary of the pitch as one European project's own material puts it [33]. Your job is to know which half of that sentence you are being sold. The reference-architecture view says costs run across development, operation, evolution and maintenance [19]; the six lines below follow that lifecycle from inception to disposal, roughly ordered by how often each is the surprise.

1. Getting the data out, and keeping it correct. Connectivity, protocol translation, timestamp discipline, units, and the context that says *which* vessel a concentration reading belongs to. This is your home turf and it is consistently underestimated. In the dosing rig's value case it is most of C_{build}, and it is the line the pilot reports itemise most concretely – one records the operating cost of radio-frequency identification (RFID) tags per can, and the licence terms for running its simulation software in the cloud [3].

2. Building and – worse – maintaining the model. The first version of a model is a project. Keeping it valid as the PT wears, gets repaired, or has a component swapped is a *permanent* cost. The twin has to track both continuous change, such as wear and tear, and discontinuous change, such as a component replaced after a failure [34]. The pilot reports reach the same place from the field: their final step, keeping the twin synchronised with the physical world and its knowledge managed, is the one they say will dominate the twin's life after the pilot ends [3]. This is the C_{run} line, and it is the cost most business cases omit entirely.

3. People who can do this. A systematic review of process-industry adoption names the lack of specialists and expertise as a barrier and workforce upskilling as the corresponding enabler [35]. A perspective on scaling twins puts the structural version of the same point: twins today are labour-intensive, highly specialised, *artisanal* products, and the barrier to wider adoption is that the methods do not transfer across disciplines [36].

4. Compute and licences. Real, and the least alarming of the six. The same process-industry review lists lower and decreasing hardware costs among the enablers rather than the barriers [35], and a construction-engineering study reports that running a twin atop managed cloud services made it both cheaper to build and cheaper to keep running [37]. Simulator reuse – third-party or customer-supplied simulators instead of writing your own – is a documented cost-reduction tactic [32].

5. Bespoke integration, over and over. The structural problem of the field: most twins are hand-built one-offs, and the perspective above argues that scaling them from artisanal to industrial requires the barriers to adoption to come down [36]. Every hour you spend re-inventing an interface that a standard already defines is cost with no value attached. Standardised reference architectures such as ISO 23247, the International Organization for Standardization's

reference architecture for twins in manufacturing, exist to stop that – and an analysis of documented architectures finds them only partly aligned with it, which is itself a measure of how much re-invention is going on [38].

6. Disposal. A twin outlives neither its PT nor its sponsor. When the rig is dismantled or the turbine retired, someone has to decide what happens to years of stored history, the calibrated models, and the interfaces other systems came to depend on. The pilot reports' "cradle to grave" framing [3] and the lifecycle view of evolution [34] both put decommissioning inside the twin's life, not after it. It is rarely costed at the start, and it is cheap to cost.

1.6.1 The shape of the estimate

For a first-increment twin, a serviceable rule for the kick-off meeting is this: **expect integration and data work to be the largest single line, expect model maintenance to be the largest line you forgot, and expect compute to be smaller than everyone fears.** The dosing rig's estimate in the Key Question section has exactly that shape – integration dominates C_{build}, upkeep dominates C_{run}, hosting is a rounding error. Then go and measure, because a rule of thumb is not an estimate.

1.7 Application Areas

The point of this section is not the technology. It is to show you four *different value arguments*, each with a different reason the money is there. Read each one asking: what is the decision, and what does being wrong cost?

1.7.1 Aerospace and defence – the cost of being wrong is enormous

Two US aerospace bodies – the American Institute of Aeronautics and Astronautics (AIAA) and the Aerospace Industries Association (AIA) – jointly published a position paper laying out the industry's rationale for accelerating DT adoption, and it is candid that the significant work under way is often siloed and uncoordinated within each organisation's own view of the system [39]. The defence side frames the same territory as *digital engineering*: an integrated approach using authoritative sources of system data and models across the lifecycle, with a lineage traced to Apollo-era simulation for risk reduction [8].

- **Decision:** whether this airframe or engine can safely fly another number of cycles; whether a design change is acceptable.
- **Why it pays:** the cost of being wrong includes lives and airframes. Structural-health work in this family uses twin approaches for life prediction under unpredictable operational conditions [40].
- **What it demands of you:** credibility and traceability above all. Not speed. This is *certify* territory.

1.7.2 Rotating machinery and offshore assets – the cost of *going there* is enormous

Offshore wind and offshore platforms have a distinctive economics: the asset is remote, visiting it is expensive and weather-dependent, and a failure found early is worth far more than the same failure found late. That is why the sector is a heavy adopter of condition-based and predictive maintenance with twins as the enabler [22]. North Sea offshore platforms have had their fatigue tracked industrially with a twin approach, built up in five levels running from screening up to

tracking accumulated fatigue [40]. A vendor's digital wind farm, giving every turbine its own twin to sharpen maintenance strategy, turns up in practically every list of standard applications [41].

- **Decision:** send a vessel and a crew, or don't.
- **Why it pays:** each avoided unnecessary trip, and each avoided catastrophic-but-preventable failure, is directly costed. This is *diagnose* and *predict* territory, and G_r – the sailing – is long, which is exactly why shortening G_d is worth so much.
- **What it demands of you:** intermittent, lossy connectivity; edge processing; long-horizon data.

1.7.3 Process and discrete manufacturing – the cost of being *slightly* wrong, multiplied

Here no single error is catastrophic, but tiny per-unit losses multiply across enormous volumes. The literature is correspondingly focused on the enablers and barriers to getting twins into the plant at all [35], and on where value shows up: per-component early warning and plant-wide cost control across a turbine fleet [29], reference architectures for predictive maintenance [42], and platform selection against a specific goal such as net-zero targets [16].

It is not only for giants. One case study follows a small-to-medium manufacturer of roll-to-roll printed labels that deliberately kept its technology spend small and its risk lower still, while still protecting the return [43] – the lean-increment argument of the Key Question section showing up in the field.

- **Decision:** run, adjust, or stop; accept or reject a batch.
- **Why it pays:** volume. A 0.3% scrap reduction is a real number when the denominator is millions.
- **What it demands of you:** integration with systems that predate you, and uptime.

1.7.4 Health, cities and infrastructure – the cost is societal, and the twin is contested

Health DTs have grown fast enough that the field already needs scoping reviews to keep track of itself [14], and it is organised enough to have published a position paper on the barriers slowing adoption and on what a shared ecosystem would need [44]. Smart-city work maps twin value onto four concrete municipal problems – running energy systems, keeping mobility sustainable, managing water resources, and responding to disasters – and onto evidence-based decision making for a community [15].

- **Decision:** a clinical or a policy one.
- **Why it pays:** outcomes, and cost avoidance at population scale.
- **What it demands of you:** governance, privacy, and an unusually high standard for explaining what the model does. This is the family where "the model said so" is least acceptable as an answer, and where *control* is least likely to be permitted.

1.7.5 The common thread

In all four, the twin is paid for because **a decision was being made too late, too blind, or too expensively** – and the twin moves it. None of the four is paid for because a virtual replica is interesting.

1.8 Summary

Back to the objectives this chapter opened with.

1. **Decision first.** Buy a twin to make one decision better. Name the decision, the decider, the cadence, and the cost of being wrong; compress them into one value metric with a measured baseline M_0; compute the benefit from Equation 1.1 and the payback from Equation 1.2. Verify a defined financial benefit quickly and let it fund the next increment [18].
2. **Decision latency is what twins usually sell**, and only half of it is yours: Equation 1.3 splits it into a detection gap engineering can shorten and a response gap the organisation owns [4].
3. **Six value patterns.** *Monitor, diagnose, predict, decide, control, certify*. Each rung typically needs the ones below and costs more. A person acts for the first four; at *control* the twin itself acts, and at *certify* it stands in for the PT altogether. *Diagnose* – built entirely around the residual from Equation 1.4 – is very often the right first increment.
4. **Cost structure.** Data integration is the biggest line, model maintenance is the biggest forgotten line [34], expertise is the most-reported barrier [35], compute is smaller than people fear, bespoke one-off integration is the field's structural cost problem [36], and disposal is the line nobody costs.
5. **Reading the evidence.** Market forecasts are large, inconsistent and analyst-sourced; adoption lags them; sector reports beat global totals. Four deployment families pay for twins for four different reasons, and all four reduce to moving a decision earlier, cheaper or safer. Eight conditions say *don't build it*.
6. **A complete value case.** For the dosing rig, the obvious metric (reagent) was worth a few hundred euros a year; the real metric (scrapped batches) was worth EUR 29,600; the right pattern was *diagnose*, not the flashier closed loop; the fidelity requirement fell out of the metric; payback was about nine months; and one afternoon of querying stored history can falsify the whole thing before you write any code.

The one sentence to carry forward: **fidelity is a requirement derived from a value metric, not a virtue pursued for its own sake.**

1.9 Exercises

Solutions or hints follow the exercises; each says which. The objective each exercise serves is given in brackets.

Exercise 1 (Objective 1, easy). For each request, write the value metric behind it with a unit, a denominator and a stated measurement method: (a) "improve building energy performance"; (b) "reduce risk in the substation"; (c) "make the production line smarter".

Exercise 2 (Objective 2, easy). Classify each system into one of the six patterns, say who acts on its output, and state the *next* rung up the ladder and what it would additionally require: (a) a dashboard showing live bearing temperature across forty pumps; (b) a system that flags a pump whose vibration deviates from its model; (c) a system that estimates each pump's remaining useful life; (d) a system that chooses which of forty pumps to service during a six-hour maintenance window; (e) a system that throttles a pump automatically when its model predicts cavitation.

Exercise 3 (Objective 1, medium). Finish Example 2. Compute P for the pump fleet from Equation 1.2, then name G_{r} for that decision and state the one organisational fact that would make the case fail even at that payback.

Exercise 4 (Objective 4, medium). A customer asks for a twin that predicts equipment failure two weeks ahead. Their maintenance schedule is fixed quarterly and cannot be changed

within a quarter. Write the two-sentence reply you would give in the meeting, and state what you would propose instead.

Exercise 5 (Objective 4, medium). A vendor's slide says "the digital twin market will reach USD 137 billion by 2030". Using the Market Forecast section, write three questions you would ask before repeating the number, and say what each answer would change.

Exercise 6 (Objective 3, medium). Rank the six cost lines of the Major Cost Contributors section for (a) the dosing rig and (b) the pump fleet, and say for each which line you would expect the sponsor to have left out of their own estimate.

Exercise 7 (Objectives 1 and 5, hard). Do Example 3, the municipal park, in full: decision, metric, baseline, target, pattern, fidelity requirement, cost, payback, and the test that could kill it. Then write the single paragraph you would put at the top of the proposal for a reader who will read only that paragraph.

Exercise 8 (Objective 3, hard, open-ended). The cost section claims model *maintenance* is the most-forgotten cost. Take a software system you have personally maintained and estimate the ratio of its total maintenance cost to its original build cost. Then argue whether a physics model's ratio should be higher or lower, and why. There is no correct answer; there is a correct *form* of answer, which names the drivers of change on both sides [34].

1.9.1 Solutions and hints

Exercise 1 (solution). (a) "kWh per square metre per heating-degree-day, from the building meter, measured over one heating season." (b) "Unplanned outage minutes per year, from the Supervisory Control and Data Acquisition (SCADA) event log" – note that "risk" is not a metric until you say risk *of what*, measured *how*. (c) There is no answer, and that is the point: "smarter" names no decision. The correct response is the four questions of the Key Question section.

Exercise 2 (solution). (a) *monitor*, a person acts; next is *diagnose*, which additionally requires a model of expected temperature and a residual. (b) *diagnose*, a person acts; next is *predict*, which additionally requires a forecast horizon and an uncertainty estimate [27]. (c) *predict*, a person acts; next is *decide*, which additionally requires running many scenarios fast [28]. (d) *decide*, a person still chooses; next is *control*, which additionally requires a safety argument for acting unattended. (e) *control*, the twin acts and a person supervises.

Exercise 3 (solution). $P = 12 \times \frac{90{,}000+30{,}000}{120{,}000} = 12$ months. G_{r} is the planner's morning round plus the crew's travel – about a day. The fact that kills the case is a maintenance contract that fixes visits by calendar: then G_{d} shortens and nothing else moves, which is condition 3 of "When the right answer is no".

Exercise 4 (hint). Condition 3 of "When the right answer is no", and the warning under *predict*. A good reply names the mismatch without blaming the customer, and proposes the rung below: *diagnose*, which is actionable within their existing quarterly cycle because it tells them what to put *on* the next quarter's list.

Exercise 5 (hint). Who measured it, and what did they count as a digital twin? What was the adoption figure in the same report? Which sector figure, rather than the global total, applies to this customer? Each answer moves the number from evidence of size to evidence of expectation.

Exercise 6 (hint). For the dosing rig, expect the sponsor to have costed compute and forgotten model upkeep; for the fleet, expect three pump vintages to make integration the surprise and the vibration models the forgotten line.

Exercise 7 (hint). There is no solution, but there is a test. If your paragraph contains a number without a source, it is not finished. Start from the water meter and the turf-replacement invoices: those are the two baselines that already exist.

Exercise 8 (open). Grade it on whether the answer names a driver of change on each side: requirements churn and dependency updates for the software; wear, repair, swapped components and PT reconfiguration for the physics model [34].

1.10 Further Reading

- *A full book on engineering cyber-physical twins*, the closest fit to this book's angle: the opening chapter of an edited volume [45], its chapter of worked case studies [46], and its chapter on advanced services and what-if analysis [30], all in [47].
- *Where the concept came from:* [5] for the 2002 origin, [6] for the naming history and the NASA definition, [7] for the Apollo lineage, and [48] for the question of whether twins are an evolution of modelling and simulation or something new.
- *Surveys with breadth:* [10] on enabling technologies and trends, [41] on tools, [9] on the business-model angle, [49] on the networking and Internet-of-Things side, [21] on the concept's evolution, and [12] on moving from twins to twinning systems.
- *Value and adoption specifically:* [18] on the lean approach this chapter's method is built on, [35] on enablers and barriers, [2] on being realistic about return on investment, [4] on why the process has to change too, [19] for the business viewpoint of a reference architecture, [3] and [33] on pilot experience, and [43] for a small-manufacturer case.
- *Autonomy and the levels people will quote at you:* [50] for the 5C architecture, [24] for the descriptive-to-autonomous maturity ladder, [51] for the model / shadow / twin cut by data flow, and [52] for the monitor-analyse-plan-execute over a shared knowledge base (MAPE-K) loop behind autonomic twins.
- *Sector deep-dives:* [13] for oil and gas, [14] and [44] for health, [15] for smart cities, [39] and [8] for aerospace and defence, [40] for offshore structural health, and [16] for net-zero manufacturing platforms.
- *Small, reproducible exemplars* of the kind the worked example imitates: [53] presents a greenhouse twin as a community exemplar, [54] covers lifecycle management on a comparable set-up, [55] is a field-scale agricultural twin on an open platform, [56] catalogues architecture patterns using agriculture cases, and [31] is a small-scale reproducible exemplar built around a robot vehicle.
- *Consulted but not drawn on above,* and worth knowing they exist: [38] on ISO 23247, [25], [26] and [42] on predictive maintenance, [20] on the executable twin and model order reduction, [22] and [23] on diagnostic twins, [57] on what a twin has to *know* to be trusted, [29] on industrial applications, [27] and [11] on modelling and uncertainty, [17] as an accessible overview, [32] on continuous integration for cyber-physical systems, [37] on running a twin on managed cloud services, [36] on scaling, and [34] on evolution as the core of twin engineering. Further afield, on architecture and reference models: [58] on a cloud-based reference architecture for cyber-physical systems, [59] on architecting twins in general, and [60, 61] on a unifying reference model for cyber-physical twins. On engineering-design and manufacturing use: [62, 63] on developing and leveraging twins in engineering design, [64] on model-based engineering of multi-purpose manufacturing twins, [65] on a digital shadow for robot motion control, [66] on reduced-order models for predictive twins, and [67] on a cyber-physical maintenance case study. On lifecycle, testing and data: [68] alongside its companion entry above on lifecycle management, [69] on microservice-based multi-level testing, [70] on integrating past, present and future data into one model, and [71] on Asset Administration Shell communication. On security and health: [72] on twin cybersecurity in Industry 4.0, [73] as a second scoping review alongside the one cited above, and [74] on a molecular twin for cancer-outcome prediction. And [75] is the edited volume that [15]'s smart-cities chapter, cited above, appears in; and [76]

applies pretrained language models to forecasting when-it-happens outcomes for twinned equipment.

Chapter 2

Plant Demonstrator: A Case Study

2.1 Before You Start

What you are assumed to know. You write software for a living, you are comfortable with HTTP APIs, JSON, a time-series or relational database, and reading Python you did not write, and you have seen a system that talks to hardware. Two terms are used throughout: the **physical twin (PT)** is the real system being twinned, and the **digital twin (DT)** is the software kept in step with it, which answers questions about it and, in time, controls it.

What you are not assumed to know. Anything about electronics beyond plugging a cable into a socket, anything about serial buses, anything about soil or plants beyond having kept one alive or failed to, and anything about control. Where this chapter needs a term from one of those fields it defines it at first use and then keeps using it.

What this chapter covers. This book needs one concrete physical system that every later chapter can point at, small enough to sit on a desk and cheap enough that a reader could build it. This chapter introduces that system: a potted plant, watered by a small pump, under a Raspberry Pi that reads soil, air and light sensors and exposes everything over an HTTP interface. It is a real rig that others have built and documented, not an invented one; a pointer to its build documentation is in Further Reading, and this chapter stays at the level of the overview. It describes the rig as a block diagram, identifies which parts are sensors and which are actuators, walks up the four protocols that carry a soil reading from a probe to a JSON document, draws the line where the PT's software boundary sits, works two watering strategies and the pump calibration both depend on, and closes by asking what a DT of a plant would be for and what it would have to measure. Along the way it writes down a small mathematical model of each side: the pot, pump and controller below the boundary, and the twin that predicts and checks above it.

What this chapter deliberately does not cover. How to assemble the rig or which parts to order: this is an overview, and the build documentation is where those details live. Plant physiology or soil science beyond the minimum needed to motivate a metric. Electronics below the block-diagram level. And the DT as software – its architecture, how its model is calibrated, how that calibration is kept honest – which is the subject of the chapters that follow. This chapter ends where the DT's equations are on paper and its code is not.

Learning objectives. By the end of this chapter you will be able to:

1. **Describe** the plant controller as three groups of blocks, **classify** each block as a sensor, an actuator or neither, and **name** which of its two buses each one hangs on.
2. **Trace** one soil-moisture reading from the probe to a JSON document in a browser, naming the protocol in use at each hop and what each protocol adds.

3. **Explain** why a DT should depend on the controller's REST API and on nothing beneath it, and **relate** that boundary to the physical adapter of a general-purpose DT framework.
4. **Compute** a pump's calibration from a table of measurements, **convert** a dose in millilitres into a relay-on time, and **state** what goes wrong when calibration is skipped or stale.
5. **Compare** a fixed-schedule watering strategy with a target-moisture strategy by how long a plant stays too dry before anything acts, and **write** the REST call that switches the rig from one to the other.
6. **Propose** three or four metrics a plant DT should monitor and control, each with a unit and a reason.
7. **Write down** the equations of the physical twin – drying, dose response, pump time, delivery – and of the digital twin – dose rule, prediction, residual – and **run** them by hand across one check interval.

Every section that follows serves at least one of those seven, and the exercises at the end say which.

2.2 Physical Twin

The PT is *the real system being twinned*. For this book that system is a plant in a pot. Start with the parts you can see.

2.2.1 What is on the desk

On the desk there is a potted plant with a plastic tube running into its soil and a small probe pushed into the soil beside it. The tube comes from a submersible pump sitting in a water tank. Next to the pot is a Raspberry Pi 5 with its power supply, a USB dongle, a small relay board, a second power supply for the pumps, and a chain of thumbnail-sized sensor boards joined by short cables. That is the whole PT. This chapter says "the controller" for the Raspberry Pi and the software on it, and "the rig" or "the PT" for the whole thing including plant, pot, tank and tubing.

Two design goals shape everything below. The rig is built from **off-the-shelf parts**, so that it can be rebuilt on another desk without any electronics design. And it is meant as a **physical twin test kit for digital-twin work**: each plant can be watered on a schedule or on demand, and its parameters, along with those of its surroundings, are continuously measured, recorded, and exposed over an HTTP interface for a DT to consume. The first goal is why the parts list is short; the second is why the software boundary in the Physical Interface section exists at all.

It is not the only rig of its kind, and knowing the neighbours helps you see what is a design choice and what is a convention. GreenhouseDT, a community exemplar from the University of Oslo, twins a small greenhouse where plants, sensors and irrigation pumps together make up the physical rig, run by Raspberry Pi minicomputers, and its authors note that sensors and actuators have to be calibrated for the individual asset [53]. A study of reproducibility in DT research, built around a small robot car, argues that research twins should be rebuildable by others, and describes ocean-observation platforms whose sensors and actuators sit on a fieldbus controller over an RS-485 bus with a Raspberry Pi exchanging data and commands with it [31]. The rig in this chapter is a member of that family: cheap single-board computer, commodity sensors, one serial bus for the things that need distance, a database, an API.

2.2.2 The block diagram

Figure 2.1 is the rig as three groups and the plant they all end at. Read it top to bottom.

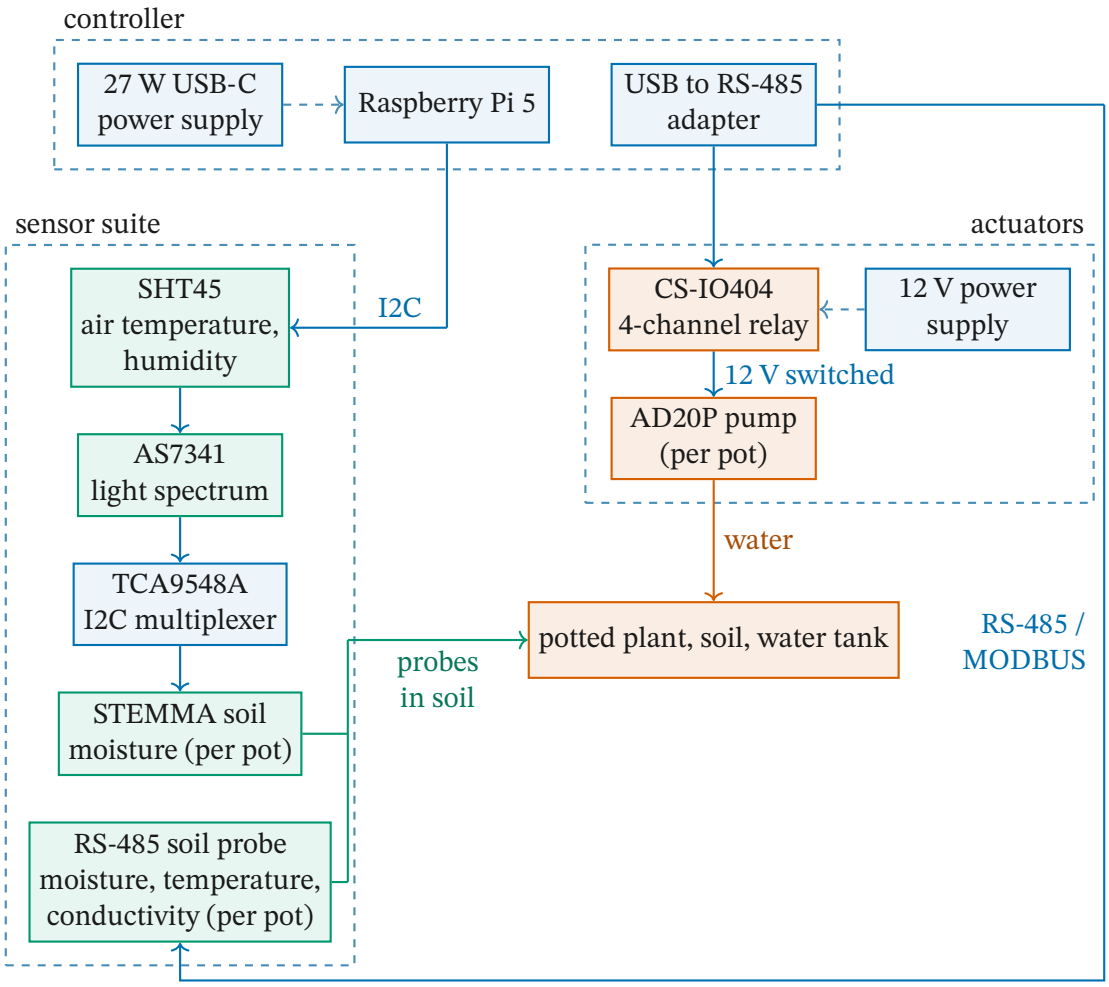


Figure 2.1: The plant controller as three groups – controller, sensor suite, actuators – and the potted plant every path ends at. Green blocks are sensors, orange blocks are actuators, blue blocks are neither. The I2C chain runs down the left, the RS-485 bus down the right.

The **controller** group is the Raspberry Pi 5, its 27 W USB-C power supply, and a USB-to-RS-485 adapter. The Raspberry Pi runs the controller software and is the only computer in the rig. Its wireless network connection is how anything outside the desk reaches it.

The **sensor suite** is everything that turns a physical quantity into a number. Two sensors watch the surroundings, which the software calls the greenhouse even when the surroundings are an office: an SHT45 module measures air temperature and relative humidity, and an AS7341 module measures light as a spectrum across nine wavelength bands from violet to near infrared. Per pot there is one of two soil probes. The cheaper option is a capacitive STEMMA soil-moisture sensor, which reports moisture alone. The recommended option for DT work is an RS-485 soil probe that reports moisture, soil temperature and electrical conductivity from one set of tines. Because every STEMMA sensor answers to the same address, a TCA9548A multiplexer sits between the chain and the soil sensors so that up to seven of them can share one bus.

The **actuators** are everything the controller can command to change the world. There is exactly one kind: an AD20P submersible pump per pot, switched on and off by one channel of a CS-IO404 four-channel relay module. A 12 V power supply feeds the relay board and, through it, the pumps.

Everything else in the picture – the two power supplies, the RS-485 adapter, the multiplexer,

the cables, the tubing, the tank, the pot – is neither sensor nor actuator. That distinction is worth being strict about, because a later chapter's model of the rig has an input for every actuator and an output for every sensor and nothing else, and a part that is neither does not appear in the model at all. Table 2.1 makes the classification explicit.

Table 2.1: Every block in the rig, classified. Only the green and orange rows of the figure appear in a model of the rig; the rest is plumbing. The two power supplies are left out: they power blocks and are not themselves part of the twin's picture.

Block	Group	Sensor, actuator or neither	Bus	What it does
Raspberry Pi 5	controller	neither	–	runs the software and hosts the API
USB-to-RS-485 adapter	controller	neither	RS-485	lets the Raspberry Pi speak MODBUS
SHT45 module	sensor suite	sensor	I2C	measures air temperature and relative humidity
AS7341 module	sensor suite	sensor	I2C	measures the light spectrum in nine bands
TCA9548A multiplexer	sensor suite	neither	I2C	lets several same-address sensors share the bus
STEMMA soil sensor	sensor suite	sensor	I2C	measures soil moisture, one per pot
RS-485 soil probe	sensor suite	sensor	RS-485	measures soil moisture, temperature and conductivity, one per pot
CS-IO404 relay module	actuators	actuator	RS-485	switches 12 V to each pump
AD20P pump	actuators	actuator	–	moves water from tank to pot
tank, tubing, pot, plant	–	neither	–	the thing being twinned

Read the table down its third column: four kinds of sensor, two kinds of actuator, and four things that are neither. Now read the fourth: there are only two buses, and the RS-485 one carries both a sensor and an actuator. That is the fact to hold on to for the next section.

2.2.3 The two buses, and the power

The sensor boards along the left of Figure 2.1 are joined in a **daisy chain**: a four-wire cable runs from the Raspberry Pi's header to the SHT45, a second from the SHT45 to the AS7341, a third on to the multiplexer, and from the multiplexer one cable per soil sensor in a star. The four wires are power at 3.3 V, ground, a data line and a clock line. This is an I2C bus, and the next section says what that means; for now, note that the whole chain is powered from the Raspberry Pi and that it is short – centimetres, not metres.

The devices along the right hang on a different bus. From the RS-485 adapter run two wires, twisted together, to the relay board's `A+` and `B−` terminals; the RS-485 soil probes are spliced onto

the same two wires wherever convenient. Good practice asks for the twist, and for a 120 ohm resistor across each end of the pair, for a reason the next section gives. This bus is long – metres are fine – and it is the one on which the controller sends commands, not only reads values.

Power follows three separate paths, and keeping them separate in your head prevents the commonest wiring mistake. The Raspberry Pi and the whole I2C chain run from the 27 W supply at 3.3 V and 5 V. The RS-485 soil probes take 5 V from the RS-485 adapter's own power terminals. The relay board and the pumps take 12 V from the second supply, and each relay channel does nothing more than connect that 12 V to one pump for as long as the controller tells it to. Nothing on the I2C side ever touches 12 V.

2.2.4 What the sensors actually report

A block diagram tells you what is measured; a DT will want to know in what units and how often. Table 2.2 lists every quantity the rig produces, as the controller software labels it.

Table 2.2: The quantities the rig produces, named as the controller stores them. The `watering` row is not a sensor reading; it is the controller's own record of what it did.

Unit	Parameter, as stored	Reported as	Time between reads
greenhouse	`air_temperature`	degrees Celsius, with a stated confidence of plus or minus 0.2 degrees at 95 per cent	configurable
greenhouse	`air_humidity`	per cent relative humidity	configurable
greenhouse	`light_level_415nm` through `light_level_910nm`	photon flux, photons per second, nine bands	15 s
plant (STEMMA)	`moisture`	per cent of the sensor's full scale	configurable, 10 s in the shipped example
plant (RS-485 probe)	`soil_moisture`	per cent	configurable
plant (RS-485 probe)	`soil_temperature`	degrees Celsius	configurable
plant (RS-485 probe)	`soil_electrical_conductivity`	microsiemens per centimetre	configurable
plant	`watering`	one event per pump run, with its dose in millilitres	on each dose

Two rows deserve a second look. The STEMMA sensor's "per cent" is not a volumetric water content: the software reads a ten-bit number between 0 and 1023 from the probe and scales it linearly to a percentage, so a raw reading of 512 becomes $100 \times 512/1023 = 50$. The source comments say so and call for a calibrated transform to replace it. Treat that per cent as a relative scale – wetter is higher – and nothing more, until the chapter on calibration gives you a reason to trust it further. And the `watering` row is different in kind from every other: it is not something

the rig observed but something the rig *did*. A DT that forgets the difference will build a model of the pot from a column that is really a model of the pump schedule.

The "time between reads" column is a design decision, not a property of the hardware. The chapter of *The Engineering of Digital Twins* that surveys sensing and communication for twins puts the trade-off plainly: a sampling frequency chosen for the worst case generates far more data than needed, while one chosen for the typical case can become inadequate when the environment changes, and the choice of sensors and their characteristics has to be made early, at design time [77]. For soil in a pot, which changes over hours, a read every ten seconds is generous. For a greenhouse whose door opens, fifteen seconds on the light sensor is about right.

2.2.5 A mathematical model of the physical twin

Everything above is a description. A DT needs the PT in a second form: a handful of equations that say how the numbers the rig reports change with time and with what the controller does. This subsection writes that model down, in the order a reader can run it by hand – state and parameters first, then the pot, then the controller and the pump it drives. It is deliberately the simplest model that closes the loop; the calibration chapter replaces its constants with fitted ones, and its straight lines with curves where the data demand it.

State, parameters and initial condition. The model tracks one quantity that changes, the soil moisture m, on the same per-cent scale the probe reports. Everything else is a parameter: a number that holds still over a day and that a calibration sets. Table 2.3 lists them.

Table 2.3: The physical twin's state and parameters. There are two pairs of pump constants because the controller acts on the pair it has stored while the pump obeys the pair it really has; the two agree right after a calibration and drift apart afterwards.

Symbol	Meaning	Unit	Where it lives
m	soil moisture, the state; needs a starting value $m(0)$	per cent	the pot, read by the probe
d	drying rate: what the pot loses in an hour with no dose	percentage points per hour	the pot
g	dose response: what one millilitre adds to the reading	percentage points per millilitre	the pot
a, b	the calibration slope and offset the controller has stored	seconds per millilitre; seconds	the controller's configuration
a_{true}, b_{true}	the slope and offset the pump really has today	seconds per millilitre; seconds	the pump, tank and tubing

The pot dries. With no dose, moisture falls at the drying rate. After h hours,

$$m(t + h) = m(t) - d\,h \tag{2.1}$$

Equation 2.1 is the whole of the pot's physics as this chapter needs it. A constant d is a simplification a reader can see through at once – a pot dries faster on a hot bright afternoon than at night, and the greenhouse unit measures exactly the quantities that would let d vary – but it

is the right first model, because it is the one whose parameter the rig's own data can recover, as the Plant Digital Twin section shows.

A dose raises it. A volume of water V_{del} that actually arrives in the pot lifts the reading by g points per millilitre:

$$m_{\text{after}} = m_{\text{before}} + g\,V_{\text{del}} \tag{2.2}$$

The subscript on V_{del} matters. Equation 2.2 takes the millilitres that reach the soil, not the millilitres a schedule asked for; the next two equations connect the two, and the gap between them is where a stale pump hides.

The controller turns a dose into a time. The relay knows only *on* and *off*, so the controller converts a requested dose V into a pump time t with a straight line through the constants it has stored:

$$t = a\,V + b \tag{2.3}$$

The slope a is the reciprocal of the pump's flow rate; the offset b is the time the pump runs before the first drop arrives, spent lifting water up the tube. Where those two numbers come from, and how to read them, is the Watering Strategies section's worked example; here they are the controller's belief about its pump.

The pump delivers by its own constants. Held on for t seconds, the pump delivers whatever its real slope and offset say, which is Equation 2.3 rearranged with the true pair in place of the stored one:

$$V_{\text{del}} = (t - b_{\text{true}})/a_{\text{true}} \tag{2.4}$$

When the stored constants equal the true ones, Equation 2.3 and Equation 2.4 undo each other and the pot receives exactly the dose that was asked for. When they do not, every dose is short or long by an amount that nothing in the controller's own records shows.

Running it. Those four equations and a starting value $m(0)$ are the whole physical twin on paper. To run it for a day, step the clock an hour at a time: read m; deliver any dose due at that hour through Equation 2.3 and Equation 2.4 and apply Equation 2.2; then dry for an hour by Equation 2.1. The Watering Strategies section supplies the doses – a fixed schedule, or a rule that reads m – and the Plant Digital Twin section puts a model of the same shape on the other side of the API and asks what happens when the two disagree.

2.3 Low-level Protocols

Objective 2 asks you to follow one soil-moisture reading from the tines of a probe to a JSON document in a browser. Four protocols carry it, and each one adds something the one below could not. Take them in order.

2.3.1 I2C: many chips, two wires, one master

I2C (Inter-Integrated Circuit) is the bus for the sensors sitting next to the controller. It has two signal wires – **SDA** carries data, **SCL** carries a clock – plus power and ground, which is why every cable in the daisy chain has four conductors. The Raspberry Pi is the only **master**: it starts every exchange. Every other chip is a **peripheral** with a fixed **address**, and an exchange begins with the master calling out an address so that exactly one chip answers.

The address is the whole idea, and Figure 2.2 shows it for a rig with three pots. The air sensor and the light sensor each have an address of their own, so they can share the wire. The three

soil sensors do not: every sensor of that kind leaves the factory with the same address, so on one wire all three would answer at once. The multiplexer solves that. It has an address of its own and eight numbered ports; the controller first tells the multiplexer which port to open, then addresses the soil sensor behind it. "Pot 2's moisture" is therefore not one address but a pair – the multiplexer's port, then the sensor's address – and that pair is what the controller's configuration records for each pot. The numbers in the figure are the chips' defaults; what matters is not their values but the rule they obey: every device the master wants to reach must have a name no other device on that wire shares.

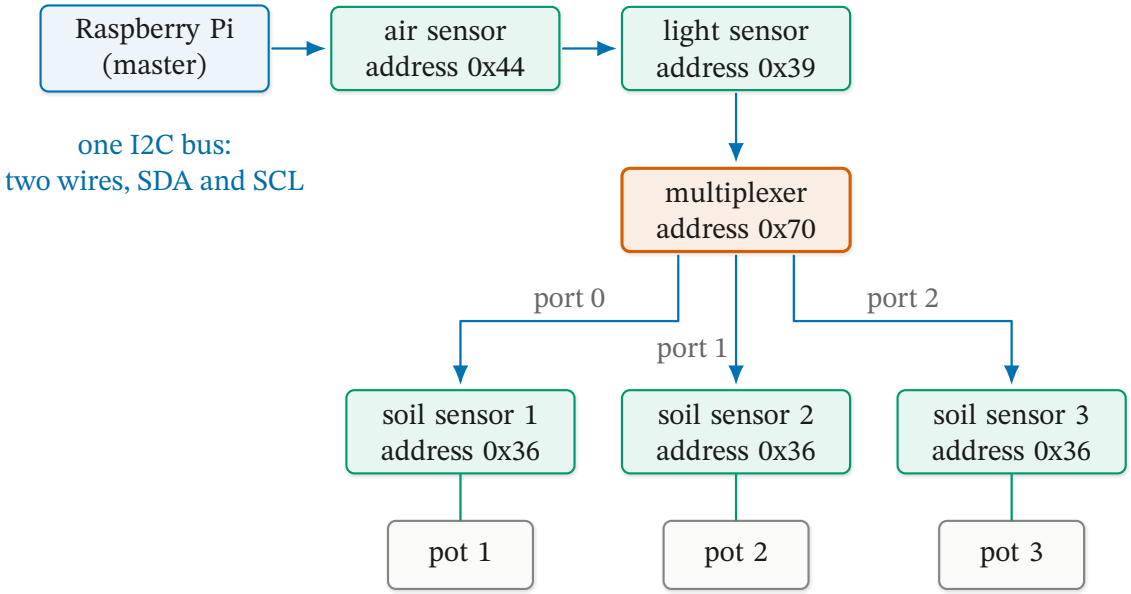


Figure 2.2: The I2C chain for three pots. Each device the controller talks to needs an address no other device on the wire shares; the three identical soil sensors get theirs by sitting behind different ports of the multiplexer.

I2C is built into nearly every small sensor chip, so the controller's Python code never touches the bus directly: a per-chip library turns "read the air sensor" into a method call. Its one weakness is distance – centimetres, not metres – which is what the second bus is for.

2.3.2 MODBUS RTU over RS-485: distance, and commands

The second bus reaches the relay board and the soil probes in the pots, which may be metres away, and it is the one bus on which the controller gives orders as well as asking questions. Two standards are stacked here, and it pays to keep them apart.

RS-485 is the wire. It is an electrical standard for a **differential** pair: two twisted wires carry the same signal with opposite polarity, so noise picked up along the cable affects both equally and cancels at the receiver. That is what buys tens of metres where I2C manages centimetres; it is why the pair is twisted, and why a 120 ohm resistor across each end of a long bus is good practice – it stops the signal reflecting back from the ends. Many devices share the one pair, and at any moment one talks and the rest listen.

MODBUS is the language spoken on the wire. It is a request-and-reply protocol from industrial automation, and it comes in three variants – TCP/IP, RTU and ASCII – of which RTU is the one for serial lines, with one master and many slaves [25]. It is old, small and everywhere: the ISO/IEC reference architecture for the Internet of Things (IoT) lists MODBUS beside Ethernet and Wi-Fi among the data-transferring capabilities an IoT system may be built on [78]. The con-

troller is the client. Each frame it sends names a **device id**, a function, and an address inside that device; the one device with that id acts and answers, and nobody else speaks.

MODBUS names the data inside a device in two ways, and the rig uses both:

- A **coil** is one on/off output. The relay module's four channels are coils 0 to 3, and "run pump 2" is a write of the value `true` to coil 1 at the relay module's device id.
- A **holding register** is one sixteen-bit number. Each soil probe exposes moisture, temperature and conductivity in three consecutive registers, which the controller reads and scales.

Figure 2.3 shows the same three-pot rig from this bus's point of view. Every device on the wire has a device id no other device shares; inside a device, coils and registers are numbered. Two probes with the same id would both answer, exactly as two identical soil sensors did on I2C – and here there is no multiplexer to rescue them, so a new probe's id is changed once, at setup, before it joins the bus. Again the numbers are illustrative; the rule is that an address is unique on its wire.

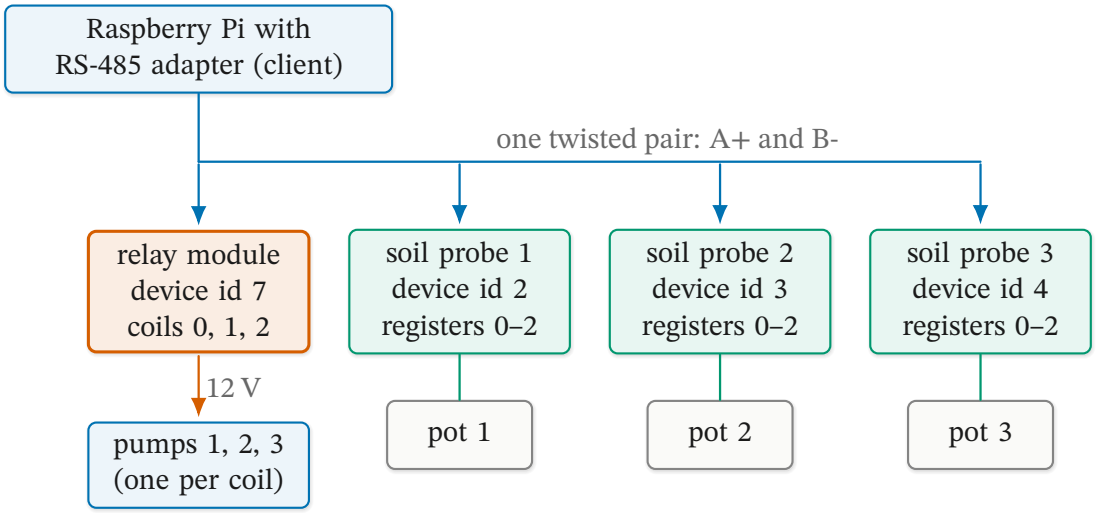


Figure 2.3: The RS-485 bus for three pots. One cable and four devices, each with a device id of its own; inside the relay module the three pumps are coils 0 to 2, and inside each probe the three quantities are registers 0 to 2.

The distinction to carry forward: on I2C the controller only *reads*. On MODBUS it reads registers *and writes coils*, and writing a coil is the single point at which software in this rig changes the physical world.

2.3.3 InfluxDB: readings become a time series

Every reading and every watering event ends up in a database on the Raspberry Pi, and the database is **InfluxDB 3**, a time-series database. A time-series database is one whose rows are timestamped values and whose queries are about time: the last hour, since Tuesday, the most recent hundred. The foundational chapter of *The Engineering of Digital Twins* describes InfluxDB as an open-source time-series database designed to handle large volumes of timestamped data, which is what makes it a common choice for monitoring and IoT work [79], and the same book's tutorial on an incubator twin uses one in exactly the rig's role: a Python service listens to the traffic between the controller and its low-level driver, then writes those messages into InfluxDB [28]. GreenhouseDT does the same, storing its sensor data in a time-series database the twin can query [53].

The storage layout is simple and worth knowing, because a DT will read it through the API. Every datum becomes a **point** with four parts: a **measurement** name, which is InfluxDB's word for a table; **tags**, which are indexed strings; **fields**, which are the values; and a timestamp. The controller keeps one measurement per unit and parameter, named `<unit>_<parameter>`, so the moisture reading for the plant named `plant_1` lands in `plant_1_moisture`, with the value, its units and, where the sensor states one, a confidence as its fields. A watering event goes to `plant_1_watering` with the dose in millilitres as its value.

Reading it back is a query in InfluxDB 3's SQL dialect, and the controller issues one shape of query only:

```
SELECT * FROM plant_1_moisture
WHERE time > '2026-09-01T00:00:00'
ORDER BY time DESC
LIMIT 100
```

The database keeps a **retention period** of seven days: older points are dropped, so the controller's memory card does not fill. That single setting has a consequence for a DT that the rest of this book will keep returning to – anything that wants more than a week of history has to copy it out before it goes.

2.3.4 HTTP: the reading leaves the desk

The last hop is the one you already know. **HTTP** (Hypertext Transfer Protocol) is the request-and-reply protocol of the web: a client opens a connection, sends a request naming a **method** (`GET` to read, `PUT` to replace) and a **resource** by its path, and the server sends back a **status code** and a body. A comparison of messaging protocols for IoT systems describes HTTP as supporting the request/response RESTful web architecture, addressing its resources by Uniform Resource Identifier in the same way the constrained-device protocol CoAP does [80], and that is the role it plays here: the controller runs an HTTP server on port 8099, and every reading the rig has ever taken is one `GET` away for anyone on the same network. The next section is about what that server exposes and why it is the most important boundary in the rig.

2.3.5 The four, side by side

Table 2.4 puts the four protocols beside each other on the axes that matter for a DT.

Table 2.4: The four protocols a soil reading passes through, from the probe's tines to a browser.

Protocol	Layer it lives at	Who talks to whom	Names things by	Reads, writes or both	Distance
I2C	wires on the desk	one master, addressed peripherals	7-bit address, plus multiplexer port	reads (in this rig)	centimetres
MODBUS RTU over RS-485	wires across the room	one client, addressed devices	device id, then coil or register number	reads registers, writes coils	tens of metres

Protocol	Layer it lives at	Who talks to whom	Names things by	Reads, writes or both	Distance
InfluxDB 3	storage on the controller	the controller software, and anything with the token	measurement, tags, time	writes points, reads by SQL	the Raspberry Pi
HTTP REST	the network	any client, one server	resource path, query parameters	`GET` reads, `PUT` replaces a schedule	anywhere on the network

Read the table down the fifth column. The only place in the entire stack where something is written *towards* the plant is the coil write on MODBUS, and the only place where an outsider can cause that write is the one `PUT` in the last row. Everything else is readings flowing away from the plant. That asymmetry is the whole design, and the next section is built on it.

2.4 Physical Interface

Objective 3 is a claim about software architecture, and it is worth stating before the mechanism: **a DT should talk to the PT through one software boundary, and to nothing below it.** In this rig that boundary is the HTTP REST API on port 8099. This section says what the API offers, why the boundary sits there, and what the general form of the idea looks like in a DT framework built for it.

2.4.1 Two actors

Two kinds of user meet this rig. The **implementer** has physical access to the controller and works on it through its operating system, usually over SSH: editing configuration files, running the setup utility, starting and stopping the software. The **remote connecter** – a person, or a computer running a DT – reaches the controller only over the network, through the REST API. The two never use the same door, and that separation is deliberate. The implementer can rewire a sensor, recalibrate a pump or change a database token; the remote connecter can read what has been measured and change when the plant is watered, and nothing else.

2.4.2 What the API offers

The controller's web layer is a small FastAPI application, and it publishes its own interactive documentation at `/docs`. The endpoints that matter are in Table 2.5.

Table 2.5: The REST API on port 8099. Seven endpoints read; exactly one writes.

Method and path	Returns or does	Notes
`GET /`	controller version, current time, link to `/docs`	a liveness check
`GET /sensing`	the list of units that have sensors	`greenhouse` plus each plant
`GET /sensing/{unit}`	that unit's parameters, each with units, confidence and time between reads	the machine-readable form of the quantities table
`GET /sensing/{unit}/{parameter}`	the stored measurements, newest first	`?limit=` caps the count; `?since_timestamp=` filters by time
`GET /actuation`	the list of units that have a pump	
`GET /actuation/{unit}/watering_events`	every dose delivered, newest first	same `limit` and `since_timestamp`
`GET /actuation/{unit}/show_schedule`	the current watering schedule as JSON	
`PUT /actuation/{unit}/update_schedule`	replaces the schedule; returns 204 with no body	422 if the schedule is malformed

Three details of the table repay attention. The unit and parameter names in the paths are the same strings the database uses for its measurement names, so the API is a thin, named view over the time series, not a second data model. Unknown units and parameters return `404` with the list of valid ones in the body, and a malformed timestamp returns `400` with the expected format spelled out (`YYYYMMDD-hhmmss`, with everything after the date optional), so a client can be wrong and find out why without reading the source. And the single `PUT` does not run a pump; it replaces a *schedule*, and the controller's own scheduler then runs the pump. A remote connecter cannot say "pump now"; it can only say "from now on, water at these times with these doses". The Watering Strategies section shows what can be built inside that constraint.

Here is the reading Objective 2 asked you to trace, at the end of its journey. The request

```
GET http://plant-controller.local:8099/sensing/plant_1/moisture?limit=2
```

returns a JSON array of the two most recent points, each carrying the tags and fields the Low-level Protocols section described:

```
[
  {"time": "2026-09-07T10:31:40", "physical_unit": "plant_1",
   "parameter": "moisture", "value": 41.2, "units": "%",
   "confidence": "+-1.0000e+00 at 95.00%"},
  {"time": "2026-09-07T10:31:30", "physical_unit": "plant_1",
   "parameter": "moisture", "value": 41.3, "units": "%",
   "confidence": "+-1.0000e+00 at 95.00%"}
]
```

Ten seconds apart, because that plant's sensor is configured with $T_r = 10$. A capacitance in a probe became a ten-bit number over I2C, a percentage in Python, a point in InfluxDB, and two objects in a JSON array. Nothing in that array says "I2C" or "multiplexer port 5", and that is the point.

2.4.3 Why the boundary sits there

Figure 2.4 draws the rig as a stack of layers, with the API as the line a DT is allowed to touch.

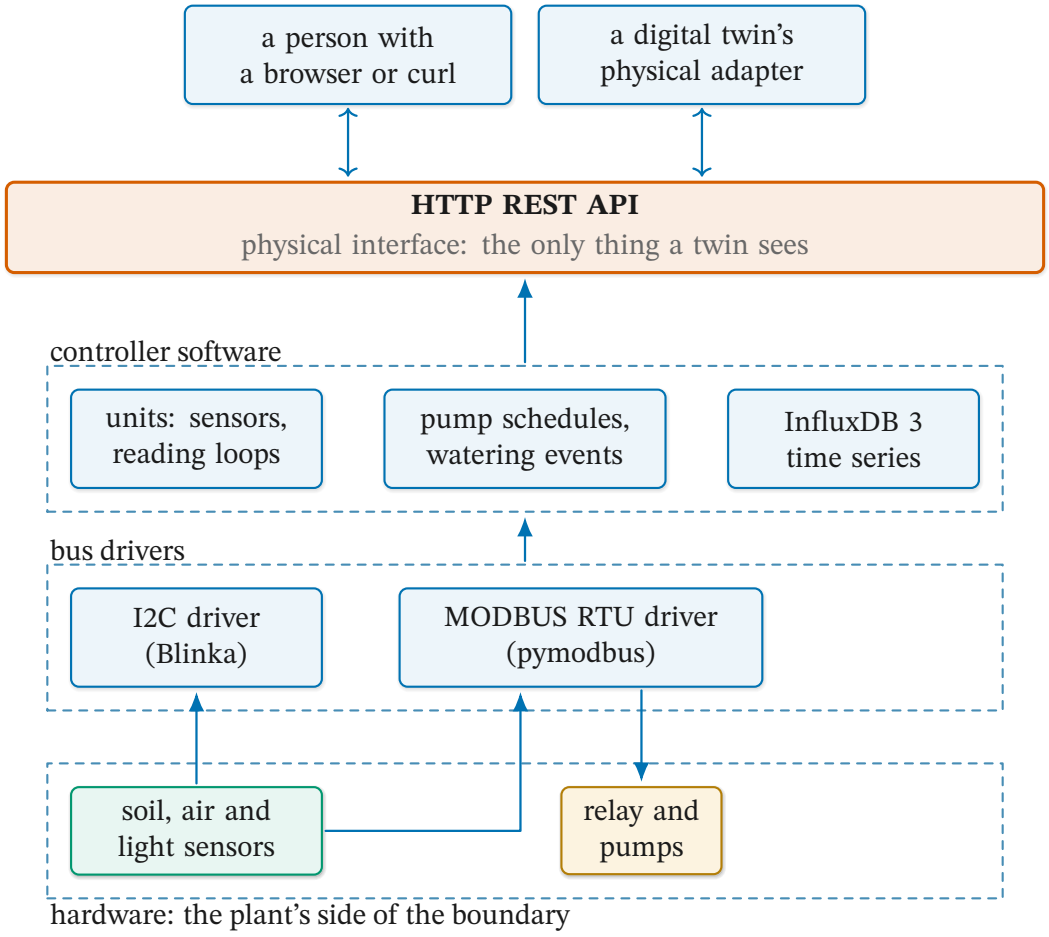


Figure 2.4: The rig as a stack. Each layer talks only to its neighbours, and a digital twin depends on exactly one of them: the REST API. Everything beneath it – controller software, bus drivers, hardware – can change without the twin noticing.

Three reasons put the line where it is, and all three are reasons you already know from ordinary software engineering, applied to a PT.

The layers beneath will change. This rig is itself the third revision of its design, and each revision swapped relay boards and sensors beneath an interface of the same shape. A DT written against `/sensing/plant_1/moisture` survives such a change; one written against a multiplexer port does not. The research literature on DT lifecycles makes the general point: a PT can change its telemetry frequency and its value ranges across the phases of its own lifecycle, and if those variations are not accounted for they complicate keeping the DT synchronised with it [81]. An API that publishes each parameter's units and time between reads, as this one does, is how a PT accounts for them.

The boundary is where responsibility changes hands. Below the line, the controller

software owns correctness: it holds the bus lock, it converts registers to degrees, it decides that a dose of 100 ml means 3.5 s of relay time. Above the line, the DT owns *interpretation*: what a moisture of 41 per cent means for this plant, whether the last dose had the expected effect. The comparison of what a model expected with what the probe reported – the **residual**, the difference between the two – is computed above the line from numbers produced below it, and neither side needs to know how the other does its job.

Safety lives below the line. The API cannot switch a relay. It can only replace a schedule, which the controller validates before accepting, and the controller alone decides how long a pump runs. A DT with a bug can water the plant at the wrong time; it cannot leave a pump running until the tank is empty and the floor is wet. Whether that limit is the right one is a design question the Watering Strategies section takes up, but that there *is* a limit, enforced on the PT's side of the line, is what makes an experimental DT safe to point at a real pot.

2.4.4 The general form: a physical adapter

Every rig draws this line somewhere, and the DT frameworks built for IoT give it a name. The **White Label Digital Twin (WLDT)** library, a Java framework for IoT twins, was designed to maximise modularity, re-usability and flexibility in mirroring physical objects into digital counterparts, and its authors describe extending a physical device's behaviour without altering it – a legacy-protocol example mirrors commercial light bulbs into digital replicas that answer external requests either by forwarding them to the devices or from a local cache, to cut response time and load on the physical objects [82]. A survey of IoT twin platforms summarises WLDT as a framework that runs at the edge or in the cloud and supports a wide range of communication protocols and normalised data formats [49].

The architectural idea behind it, which the same group later generalised as an *edge digital twin*, is that the twin is an active component that encapsulates the responsibility of talking to its physical counterpart, in a one-to-one relationship, and keeps the two in step – a process the authors call **shadowing**, supported by a module in charge of the physical layer's protocol heterogeneity [83]. In WLDT's vocabulary that module is the **physical adapter**: the one part of the twin that speaks the PT's protocols. A **digital adapter** on the other side exposes the twin to its consumers, and a **shadowing function** between them decides what the twin's state is and what a request means. An industrial case study built on the library shows the three in motion: configuration data reaches a machine's twin through its HTTP digital adapter, the shadowing function validates the request, and the physical adapter forwards it to the physical controller [84].

Set that beside Figure 2.4 and the correspondence is exact. The rig's REST API is the *physical interface*: the PT's side of the boundary. A DT's physical adapter for this rig is a small piece of code that knows seven `GET`s and one `PUT` – the twin's side of the same boundary. A later chapter gives the architecture of a DT; when it does, the physical adapter is where this chapter's material plugs in.

2.5 Watering Strategies

The rig's one actuator is a pump, and the only question a watering strategy answers is *when does it run, and for how long*. This section takes the two strategies the outline names, shows how each is expressed through the API, and then does the thing both depend on and neither can skip: calibrating the pump.

2.5.1 Strategy 1: a fixed daily schedule

The controller ships with one schedule type, called `daily`. A schedule is a JSON document with a `type` and a list of events, each with a time of day and a dose in millilitres:

```
{
  "type": "daily",
  "schedule": [
    {"time": "09:16:00", "dose": 100},
    {"time": "17:05:30", "dose": 120}
  ]
}
```

The scheduler sorts the events, sleeps until the next one, runs the pump for that dose, records a watering event, and repeats; after the last event of the day it sleeps until the first event of tomorrow. That is the entire algorithm, and it is the `daily` module's whole job.

Installing that schedule for a plant is one request:

```
PUT http://plant-controller.local:8099/actuation/plant_1/update_schedule
Content-Type: application/json

{"type": "daily", "schedule": [{"time": "09:16:00", "dose": 100},
                               {"time": "17:05:30", "dose": 120}]}
```

which returns `204 No Content` on success and `422` with a reason if, say, a dose is written as a string. The controller writes the new schedule to disk, cancels the running scheduler, and restarts it with the new events, so the change takes effect immediately and survives a reboot.

This is the commonest shape of automation there is: the pump runs on a timer, the sensor readings are stored, and the decision "how much water, and when" never reads the reading. Panel (a) of Figure 2.5 draws the loop that does not close. There is nothing wrong with it as a starting point – houseplants have been kept alive this way for as long as there have been timers – but it makes the decision once, when someone writes the schedule, and then stops deciding.

2.5.2 Strategy 2: a target moisture, checked at fixed intervals

The second strategy closes the loop, and it can be built entirely on the API's side of the line. Every T_c hours, a small program – the beginnings of a DT, or a shell script; the API does not care – fetches the latest moisture m for a plant, compares it with a **target moisture** m_t, and decides a dose:

$$V = \min\bigl(V_{\max},\ k\,(m_t - m)\bigr) \tag{2.5}$$

when $m < m_t$, and no dose when $m \geq m_t$. Here k is a **gain**, in millilitres per percentage point of moisture deficit, and $V_{\max}$ is a cap the program will never exceed in one dose. Having chosen V, the program `PUT`s a one-event schedule whose time is a minute from now, waits for the watering event to appear at `/actuation/plant_1/watering_events`, and then `PUT`s an empty schedule so nothing else happens until the next check. Panel (b) of Figure 2.5 is that loop.

Two things about Equation 2.5 are choices, not consequences. The gain k decides how hard the rule pushes: too small and the plant stays under target for days, too large and one dry reading produces a flood. The cap $V_{\max}$ is the safety argument from the Physical Interface section moved up a level: whatever k says, one check delivers at most $V_{\max}$. Choosing k well needs the dose response g of the Physical Twin section's model – how much one millilitre raises the reading in

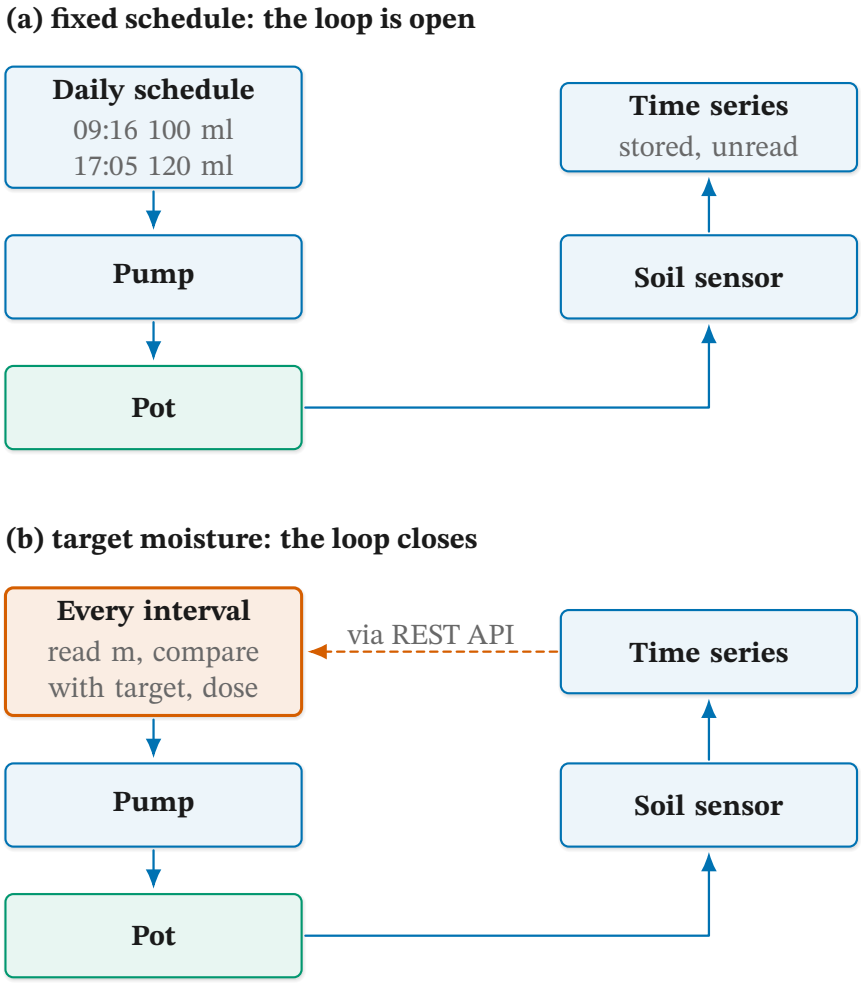


Figure 2.5: Two watering strategies on the same hardware. In (a) the schedule never reads the sensor, so the loop is open. In (b) a rule outside the controller reads the moisture through the REST API and rewrites the schedule, so the loop closes – through the API, never around it.

this pot – and fitting g is the calibration chapter's subject; for now, a conservative k and a hard cap are what make the strategy safe to run unattended.

The literature on greenhouse twins does exactly this, one level up. GreenhouseDT's authors describe behavioural adaptation as the twin calibrating the frequency or duration of running its watering pumps to how quickly the soil in the pots gets dry [53], and their work on declarative lifecycle management gives the target a reason to move: a basil plant's target moisture should shift with its growth stage, with the paper's example holding a sick plant to a lower target than a healthy one, and therefore a different controller [54]. In this chapter's terms, m_t is a parameter of the plant's state, not a constant, and the DT is the thing that knows which state the plant is in.

Compare the two strategies by how long a dry pot goes unattended, and split that time into two gaps: the **detection gap** G_d, from the pot becoming too dry to something noticing, and the **response gap** G_r, from noticing to acting. Under the fixed schedule G_d is however long it takes a person to notice a drooping leaf – days. Under the target-moisture rule it is at most T_c, and G_r is the minute the schedule takes to fire. Setting $T_c = 6$ puts the whole delay under seven hours for a fault the fixed schedule would take days to expose. The software has gone from watching the plant to acting on it, on a desk, for the price of a pump. The warning that comes with acting automatically applies too: a wrong m_t, or a probe that has slipped out of the soil and reads dry

air, will now be acted on with nobody looking, which is why $V_{\max}$ exists and why a model of what the reading *should* be matters – the Plant Digital Twin section writes one down.

2.5.3 What both strategies stand on: pump calibration

Every dose above is a number in millilitres. The relay knows nothing of millilitres; it knows *on* and *off*. Somewhere the controller has to turn "100 ml" into "hold the relay closed for this many seconds", and that conversion is **pump calibration**. Get it wrong and every strategy in this section is wrong by the same factor, silently.

The controller's model is the straight line the Physical Twin section's model already wrote down, Equation 2.3: for a dose V in millilitres the pump time t in seconds is $t = a\,V + b$, with two calibration constants per pump, a **slope** a, in seconds per millilitre, and an **offset** b, in seconds. The shipped example configuration carries $a = 0.02$ and $b = 1.5$. Read Equation 2.3 physically before reading it mathematically. The slope is the reciprocal of the pump's flow rate: at $a = 0.02$ the pump moves $1/a$, about fifty millilitres, a second. The offset is the time the pump runs before any water reaches the pot – the time to lift water up the tube and push out whatever air is in it – and it is why a dose of $V = 0$ still costs $t = b$ seconds of pumping, and why very small doses are unreliable.

Both constants depend on things the pump's datasheet cannot know: the height the water is lifted, the length and bore of the tube, how far the pump sits below the surface, whether the tube drained back after the last run. That is why GreenhouseDT's authors say sensors and actuators need calibrating for the individual asset [53], and why the controller ships a calibration procedure rather than a table of pump models.

Example 1, worked in full: calibrating one pump

The controller's setup utility runs the procedure interactively. Here is one run, with the reasoning at each step.

Step 1 – reproduce the installed geometry. The utility first asks that the pump be submerged in the full tank, connected to its relay channel, and that the outlet tube be positioned so it pumps into a measuring jug *at the same height relative to the tank as the pot*. The reasoning: b is mostly the cost of lifting water, so a calibration done with the tube lying in the sink is a calibration of a different pump.

Step 2 – measure the lift. The utility asks for the height from the water surface to the outlet, in centimetres. It uses the number to decide how long to wait between runs so the tube can drain back; for the example, the outlet sits 30 cm above the surface and the utility waits 3 s after each run.

Step 3 – run the pump for a ladder of durations and measure each volume. The utility runs the pump for 1.0 s, then 1.5 s, and so on in half-second steps up to 8.0 s, and after each run asks how many millilitres landed in the jug, read to the nearest 10 ml. Fifteen runs; the reasoning for a ladder rather than one long run is that a single point cannot separate a from b. Table 2.6 records the example.

Table 2.6: One calibration run on the example pump. The first two runs delivered nothing and are discarded by the procedure, as its source specifies.

Run	Pump time (s)	Volume measured (ml)	Kept?
1	1.0	0	no – nothing reached the jug
2	1.5	0	no – nothing reached the jug
3	2.0	20	yes

Run	Pump time (s)	Volume measured (ml)	Kept?
4	2.5	50	yes
5	3.0	70	yes
6	3.5	100	yes
7	4.0	120	yes
8	4.5	150	yes
9	5.0	180	yes
10	5.5	200	yes
11	6.0	230	yes
12	6.5	250	yes
13	7.0	280	yes
14	7.5	300	yes
15	8.0	320	yes

Read the table from the top. The first two runs delivered nothing: the pump ran, but for less time than it takes to fill the tube, and the utility discards a zero reading rather than fit to it. From run 3 on, every extra half second delivers a further 20 to 30 ml. That constant increment *is* the slope, and the point where the line would cross zero volume *is* the offset.

Step 4 – fit the line. The utility fits pump time against measured volume by least squares – the source calls `numpy.polyfit` with degree one, with volume as the independent variable, since that is the direction Equation 2.3 is used in. On the thirteen kept rows the fit gives $a = 0.0197$ and $b = 1.56$. The reasoning for fitting time on volume, rather than volume on time and inverting: the constants are used to compute a time from a dose, so the errors should be minimised in the time direction.

Step 5 – read the result physically. A slope of $a = 0.0197$ is a flow of $1/a = 51$ millilitres a second, which is about three litres a minute – far more than a pot wants, and the reason doses are seconds long. An offset of $b = 1.56$ is a second and a half of pumping before the first drop arrives, which at that flow rate is $b/a = 1.56/0.0197 = 79$ millilitres of tube and lift that every run pays for and no run delivers. Both numbers are plausible for a 30 cm lift through a metre of 8 mm tube; a slope ten times larger or an offset that came out negative would mean a kinked tube or a mis-entered reading, and the utility prints the old and new constants side by side so exactly that comparison can be made.

Step 6 – save, with a date. The utility writes the new `slope` and `offset` into the plant's configuration file together with a `calibration_date`. The date is the important part. A pump calibration goes stale: the tank level drops and the lift grows, the tube stiffens, algae narrows the bore. Keeping a model valid is the cost most twin projects forget to budget for; here it is in miniature, and the date is what lets a DT notice that the number it is trusting is three months old.

Step 7 – use it. A schedule asks for a dose of 100 ml. By Equation 2.3, the controller closes the relay for $t = 0.0197 \times 100 + 1.56 = 3.53$ seconds. The shipped constants would have given 3.50 s – close, but from a different pump on a different desk, and the 0.03 s difference is 1.5 ml on every dose, in the same direction, twice a day, for as long as nobody recalibrates.

Example 2, one step left to you: a dose from the target-moisture rule

A plant's latest reading is $m = 32$ against a target $m_{\mathrm{t}} = 40$, with gain $k = 10$ millilitres per point and cap $V_{\max} = 150$. By Equation 2.5, the deficit is eight points, so $k\,(m_{\mathrm{t}} - m) = 10 \times (40 - 32) = 80$, below the cap, and the dose is $V = 80$. **Your step:** using the constants fitted in Example 1, compute the relay time from Equation 2.3, then say what the dose and the relay time would have

been had the reading come back as $m = 21$ instead – and which of the two numbers stopped the rule from doing what the deficit asked. A check appears in the exercise solutions.

Example 3, posed as a problem: a stale calibration

Six weeks after Example 1 the tank has been topped up less often and sits lower, and the tube has been re-routed over the edge of a higher shelf. Nobody has recalibrated. Working out what happens to every dose under both strategies, which of the two strategies notices first and how, and what a DT would need to have recorded in order to raise the alarm, is the hardest exercise at the end of this chapter, and it is deliberately not worked here. The Physical Twin section's model has every equation it needs: the controller keeps using its stored a and b in Equation 2.3 while the pump obeys a changed b_{true} in Equation 2.4.

2.6 Plant Digital Twin

The chapter has so far described a PT and the software boundary a DT would use. It ends by asking the question any twin has to answer before it is worth building: what decision would a twin of this plant improve, and what would it have to measure to improve it? The full answer is the rest of the book. This section settles the motivation and the metrics.

2.6.1 Why moisture, and why minerals

A plant in a pot depends on two things the rig can influence, and the motivation for the DT is that both are easy to get wrong in both directions.

Water. Too little, and the plant wilts, stops growing, and eventually dies; too much, and the roots sit in water that has driven the air out of the soil, which kills them just as surely and more slowly. A pot is a small volume with no reserve, so the margin between the two is narrow and moves: a hot bright day dries the pot in hours that a cool dull one takes days to dry. A fixed schedule is right on average and wrong on most individual days.

Minerals. Plants take their nutrients from minerals dissolved in the soil water. In a pot those minerals come only from what was in the soil to begin with and what is added, and they leave by two routes: the plant takes them up, and every watering washes some out of the bottom of the pot. Too little and the plant starves; too much – typically from over-feeding – and the salt concentration in the soil water rises until the roots can no longer draw water against it. The rig's RS-485 probe measures the soil's **electrical conductivity (EC)**, which rises with the concentration of dissolved salts, so EC is the rig's proxy for how much mineral is in the soil water. It is a proxy, not a measurement of any one nutrient, and the chapter keeps calling it that.

Both problems are recognised targets for agricultural twins. A survey of twin platforms for the IoT lists among the uses of smart-farming twins the evaluation of soil quality by monitoring moisture, temperature, organic matter and pollutants, and the forecasting of plants' irrigation requirements to reduce fertiliser and pesticide dosages and protect groundwater [49]. A field-scale twin of an irrigation pivot on an experimental farm collects soil moisture and the levels of different elements among its inputs [55], and a catalogue of DT architecture patterns takes its worked cases from agriculture and food [56]. The pot on the desk is those systems with the hectares removed.

The two paragraphs on water and minerals state horticultural common ground rather than results from the literature. Where a later chapter needs the actual water-retention curve of a potting mix, it will have to say where it got it.

2.6.2 Four metrics

A twin needs metrics because "the plant is doing well" cannot be checked, trended or acted on, and a quantity with a unit, a denominator and a way to measure it can. Here are four the rig can compute today from what it already stores, in Table 2.7, followed by the reason for each.

Table 2.7: Four metrics a plant DT can compute from the rig's stored data. The first and last are things to hold in a band; the middle two are what a model of the pot has to explain.

Metric	Symbol	Computed from	Unit	Monitor or control?
Time in the moisture band	f_{band}	the moisture series, a lower bound m_{lo} and an upper bound m_{hi}	fraction of readings	control target
Water delivered per day	W_{day}	the watering events	millilitres per day	monitor; the cost side
Drying rate between doses	d	two moisture readings after a dose	percentage points per hour	monitor; the model's input
Conductivity drift	$EC - EC_0$	the conductivity series against its value at potting	microsiemens per centimetre	control target, slow

Time in the moisture band is the quantity whose movement anyone who cares about the plant would recognise as the point. Define a band m_{lo} to m_{hi} for the plant's current stage, count the readings that fall inside it, and divide by all readings:

$$f_{\text{band}} = n_{\text{in}}/n \tag{2.6}$$

where n_{in} is the number of readings inside the band and n the number of readings. The fixed schedule of the previous section will score poorly on Equation 2.6 on any day that is not the average day; the target-moisture rule scores better by construction; and a DT that adjusts m_{t} to the plant's stage, as the declarative-lifecycle work proposes with its lower target for a sick basil than for a healthy one [54], is trying to push it towards one. It is a fraction, so it has a denominator built in, and it is computable from a single `GET`.

Water delivered per day is the cost side, and it is the metric the rig records most reliably because it is a record of the rig's own actions. Sum the doses in the watering events for one day, $W_{\text{day}} = V_1 + V_2 + \cdots + V_j$; the fixed schedule above gives $W_{\text{day}} = 220$ every day by definition, and the target-moisture rule gives a number that varies with the weather, which is the point. On a desk the cost of water is nothing, and the metric still earns its place, because it is the input to the next one.

Drying rate between doses is the first thing a *model* of the pot has to get right. Take the moisture just after a dose has soaked in and the moisture just before the next one, and divide by the time between them:

$$d = \frac{m(t_1) - m(t_2)}{t_2 - t_1} \tag{2.7}$$

If the reading is $m = 44$ an hour after the morning dose and $m = 36$ eight hours later, then $d = (44 - 36)/8 = 1.0$ percentage point an hour. Equation 2.7 is not a control target

– you cannot set the drying rate – but it is what the model has to predict from air temperature, humidity and light, all of which the greenhouse unit already measures, and it is what the residual is computed from – the next subsection writes it down: the model says the pot should have dried by so much, the probe says it dried by this much, and the difference is the residual r. GreenhouseDT's managing subsystem relies on model-based forecasts to choose how it drives its greenhouse's actuators [53]; the drying rate is the smallest such prediction this rig supports.

Conductivity drift is the slow metric. Record the RS-485 probe's conductivity when the plant is potted, EC_0, and watch the difference $EC - EC_0$ over weeks. Falling drift says minerals are being used or washed out faster than they are replaced; rising drift says salts are accumulating. It is a control target in the sense that a person – not, in this rig, the controller – acts on it by feeding or by flushing the pot, so its delay is dominated by G_r, and the warning from the Watering Strategies section applies: a DT that shortens G_d for a decision nobody acts on has done nothing. That is a reason to keep it, not to drop it: it is the metric that will teach you, in a later chapter, the difference between monitoring and control.

2.6.3 A mathematical model of the digital twin

The metrics say what the twin watches. Its model says what it *expects*, and the difference between the two is the twin's whole advantage over a schedule. This subsection writes the DT's side down in the same order as the PT's: parameters first, then the equations, then how the two models run together across the API.

What the twin knows. The twin sits above the physical interface, so it holds only what the API gives it and what it believes: Table 2.8.

Table 2.8: The digital twin's parameters. The last two rows are its model of the pot: the same two symbols as the physical twin's model and, when the model is right, the same values. The pump constants a, b, a_{true} and b_{true} are absent on purpose; they live below the API, and the twin never sees them.

Symbol	Meaning	Unit
m_t	target moisture	per cent
k	gain: millilitres asked for per point of deficit	millilitres per percentage point
$V_{\max}$	the most the twin asks for in one dose	millilitres
T_c	time between the twin's checks	hours
d	the twin's belief about the pot's drying rate	percentage points per hour
g	the twin's belief about the pot's dose response	percentage points per millilitre

It decides. At each check the twin reads m and chooses a dose by the target-moisture rule of the Watering Strategies section, Equation 2.5: $V = \min\big(V_{\max},\ k\,(m_t - m)\big)$ when $m < m_t$, and no dose otherwise. Nothing is new here; the rule is recalled so that the model stands in one place.

It predicts. Because the twin carries the pot's two equations, it can say what the next reading should be before that reading arrives. Start from the reading m now, add what the dose should contribute by Equation 2.2, and subtract T_c hours of drying by Equation 2.1:

$$\hat{m} = m + g\,V - d\,T_c \tag{2.8}$$

$\hat{m}$ is the twin's expectation for the reading at the next check. Note what it is built from: the dose the twin *asked for*, not the volume delivered, because the twin cannot see the latter. That substitution is exactly the assumption a stale pump breaks.

It compares. When the next check arrives with the real reading m_{obs}, the twin computes the residual

$$r = \hat{m} - m_{\text{obs}} \tag{2.9}$$

A residual near zero, by Equation 2.9, says the model and the pot agree. A residual that stays positive says the pot is drier than the model expects at every check: either d is too small, or less water is arriving than was asked for. Which of the two it is can be read from *when* the shortfall appears – a wrong drying rate shows up as a larger drop *between* doses, a wrong pump as a smaller rise *at* them. That is the diagnosis the fixed schedule can never make, because it has no $\hat{m}$ to compare against.

It reports. The four metrics of Table 2.7 are the model's outputs rather than part of it. Equation 2.6 and W_{day} are read straight off the stored series, and Equation 2.7 is how the twin recovers its own parameter d from the data – the one line in this chapter where the model is fitted rather than assumed.

Running the two together. One check, box by box. The twin reads m through a `GET`; it computes r from the $\hat{m}$ it wrote down last time; it chooses V by Equation 2.5 and sends it through a `PUT`. Below the API the controller applies Equation 2.3, the pump Equation 2.4, and the pot Equation 2.2 and then Equation 2.1 for T_c hours. The twin writes down the next $\hat{m}$ by Equation 2.8 and waits. Only two numbers cross the API in that loop, a reading going up and a dose coming down. Everything the rest of the book does to the twin – a d that depends on temperature and light, a fitted g, an uncertainty on $\hat{m}$, a rule for what to do when r leaves zero – is a change to one of these equations, not to the loop.

2.6.4 What the twin would be, so far

Put the pieces together and the shape of the first increment of a plant DT is visible before a line of it is written. It sits on the twin's side of the physical interface, speaking to the rig only through the REST API. Its physical adapter knows seven `GET`s and one `PUT`. It reads moisture, conductivity and watering events; it computes Equation 2.6, W_{day}, Equation 2.7 and the conductivity drift; it holds a target moisture that depends on what it believes the plant's state to be; it closes the loop by rewriting the schedule with Equation 2.5, under a cap; and it predicts the next reading with Equation 2.8 and checks it with Equation 2.9. That is *monitoring* for all four metrics, *control* for moisture, and, through the residual, *diagnosis* of the probe and the pump. Everything the rest of the book adds – the model's calibration, its uncertainty, the architecture that holds it – plugs into that shape.

2.7 Summary

Back to the objectives this chapter opened with.

1. **The block diagram.** Three groups – controller, sensor suite, actuators – and the pot they all end at. Four kinds of sensor (air temperature and humidity, light spectrum, and per pot either a capacitive moisture sensor or a three-quantity RS-485 probe), two kinds of actuator (a pump per pot and the relay channel that switches it), and the plumbing that is neither. Two buses: a short I2C chain for the sensors near the controller, a long RS-485 bus for the relay and the probes, and on each one every device has an address no other device on that wire shares.

2. **The four protocols.** I2C addresses chips on the desk and, in this rig, only reads. MODBUS RTU over RS-485 addresses devices across the room, reads registers and writes coils – the one place software touches the world. InfluxDB 3 turns readings into a time series named `<unit>_<parameter>`, kept for seven days. HTTP carries them off the desk as JSON. The only write towards the plant that an outsider can cause is one `PUT` of a schedule.
3. **The physical interface.** The REST API on port 8099 is the PT's software boundary; a DT depends on it and on nothing beneath it, because the layers beneath change, because responsibility changes hands there, and because the safety limit is enforced on the PT's side of it. In WLDT's vocabulary the DT-side counterpart is the physical adapter, with a shadowing function and a digital adapter behind it.
4. **Calibration.** Equation 2.3 turns a dose into a relay time with a per-pump slope and offset, fitted from a ladder of timed runs; the offset is tube and lift, the slope is the reciprocal of the flow rate, both go stale, and the saved date is what lets a twin notice.
5. **Two strategies.** A fixed daily schedule leaves the loop open and the decision latency at days; a target-moisture rule, Equation 2.5, closes it through the API at a latency of one check interval, under a hard cap that is the safety argument moved up one level.
6. **Four metrics.** Time in the moisture band, Equation 2.6, is the value metric; water per day is its cost; the drying rate, Equation 2.7, is what a model must predict and what the residual is computed from; conductivity drift is the slow one a person acts on.
7. **Two models.** The physical twin on paper is four equations: the pot dries by Equation 2.1 and rises by Equation 2.2, the controller converts a dose with Equation 2.3 and its stored constants, and the pump delivers by Equation 2.4 and its real ones. The digital twin is the dose rule plus a prediction, Equation 2.8, and a residual, Equation 2.9; the residual is the one number that tells a stale pump from a hot day.

The one sentence to carry forward: **the twin sees the plant only through the API, and everything below that line is the physical twin's business.**

2.8 Exercises

Solutions or hints follow the exercises; each says which. The objective each exercise serves is given in brackets.

Exercise 1 (Objective 2, easy). A reading of soil temperature arrives in a browser as `{"value": 21.4, "units": "°C"}`. Name every protocol it passed through, in order, and for each say what the datum looked like at that hop – a register value, a Python float, a point, a JSON object – and one thing that hop added which the previous one did not have.

Exercise 2 (Objective 4, medium). Finish Example 2: compute the relay time for $V = 80$ using the fitted constants, then redo the dose and the relay time for $m = 21$, and name the number that limited the second dose.

Exercise 3 (Objective 4, medium). The calibration in Example 1 is repeated with the outlet 60 cm above the water instead of 30 cm. Say, with a reason, whether you expect a to change, whether you expect b to change, and in which direction each moves. Then say what a dose of 20 ml would deliver under the *old* constants if the geometry had changed and nobody recalibrated.

Exercise 4 (Objective 5, medium). Write, as a sequence of HTTP requests with bodies, one full cycle of the target-moisture strategy for a plant named `plant_2` with $m_t = 35$, $k = 10$ and $V_{max} = 150$, starting from a reading of $m = 30$. Include the request that confirms the dose was delivered and the request that leaves the plant quiet until the next check.

Exercise 5 (Objective 3, medium). A colleague proposes that the DT should read the soil probe's MODBUS registers directly over a second RS-485 adapter, "to avoid the latency of the

API". Write the three- sentence reply you would give, using the three reasons in the Physical Interface section, and say which one of the three is the one that should end the discussion.

Exercise 6 (Objective 6, medium). For each of the four metrics in Table 2.7, state the single `GET` request (or pair of requests) that fetches the data needed to compute it for one day, and say which metric cannot be computed for a plant fitted with the cheaper STEMMA sensor and why.

Exercise 7 (Objectives 4 and 5, hard). Do Example 3, the stale calibration, in full. State how each dose changes under both strategies when the lift grows and the tube lengthens; say which strategy's stored data shows the problem first and in which series; and list what a DT would have to have recorded, beyond the rig's own data, to raise the alarm rather than merely to have made it possible in hindsight.

Exercise 8 (Objective 2, hard, open-ended). The database keeps seven days of history. Argue for and against raising it to a year on the Raspberry Pi, and then design the smallest change – on either side of the physical interface – that would give a DT a year of history without changing the retention setting. There is no single correct answer; there is a correct form of answer, which says where the copy lives and who owns keeping it correct.

2.8.1 Solutions and hints

Exercise 1 (solution). RS-485 / MODBUS RTU: a sixteen-bit holding register holding `214`, read at the probe's device id – the hop adds an address on a shared cable. Controller software: the Python float `21.4` after division by ten, wrapped in a measurement with units and a timestamp – the hop adds units and time. InfluxDB 3: a point in `plant_1_soil_temperature` with tags and fields – the hop adds a name, an index and persistence. HTTP: a JSON object in an array returned by

```
GET /sensing/plant_1/soil_temperature
```

– the hop adds reachability from anywhere on the network and a status code that says whether the request made sense.

Exercise 2 (solution). $t = 0.0197 \times 80 + 1.56 = 3.14$ seconds. At $m = 21$ the deficit is nineteen points and the rule asks for 190 ml, which the cap reduces to $V_{\max} = 150$; the relay time is then $0.0197 \times 150 + 1.56 = 4.52$ seconds. The cap limited the dose, not the gain.

Exercise 3 (hint). The slope is about flow once water is moving, and flow through a submersible pump falls only slowly with lift, so expect a to rise a little. The offset is about filling the tube and lifting the first water, so expect b to rise clearly. Under the old constants a 20 ml dose runs the pump for $t = 0.0197 \times 20 + 1.56 = 1.95$ seconds; if the new offset is above that, the dose delivers nothing at all.

Exercise 4 (hint). Deficit five points, dose 50 ml. Four requests, in this order:

```
GET  /sensing/plant_2/soil_moisture?limit=1
PUT  /actuation/plant_2/update_schedule      (one event, a minute ahead)
GET  /actuation/plant_2/watering_events?limit=1   (repeat until it appears)
PUT  /actuation/plant_2/update_schedule      (an empty schedule list)
```

Say what your program does if the third request never returns a new event.

Exercise 5 (hint). The reason that ends the discussion is the third: below the line, two clients on one MODBUS bus with no shared lock will corrupt each other's frames, and the API's inability to switch a coil is the rig's safety property. Latency is the wrong worry; the probe is read every ten seconds regardless of who asks.

Exercise 6 (hint). Moisture and conductivity come from `/sensing/{unit}/{parameter}` with `since_timestamp` set to midnight; water per day from `/actuation/{unit}/watering_`

`events` the same way; the drying rate needs both series to find the readings around each dose. Conductivity drift needs the RS-485 probe; a STEMMA-only plant has no conductivity series to compute it from.

Exercise 7 (hint). Both strategies under-deliver by the same millilitres per dose, so the fixed schedule shows nothing in its own data – its watering events still say 100 and 120 – and the problem is visible only in the moisture series trending down. The target-moisture rule compensates by dosing more often, so its watering-event count rises while the moisture stays near target; the tell is the rising W_{day}. To raise the alarm rather than explain it afterwards, a DT needs the calibration constants and their date, and an expected drying rate to compare against.

Exercise 8 (open). Assessed on whether the answer names the memory card's size and the write load on one side and the DT's own store and its ownership on the other, and on whether the proposed copy runs on the twin's side of the physical interface.

2.9 Further Reading

- *Building the rig:* the plant-controller repository holds the build documentation this chapter deliberately leaves out – parts list, assembly guide, software installation and the REST API reference – for a reader who wants a rig of their own.
- *Comparable small twins of plants:* [53] for the GreenhouseDT exemplar and its watering-pump control, and [54] for why a target moisture should depend on the plant's stage; both are the closest published relatives of this chapter's rig.
- *Sensing, communication and storage for twins:* [77] on sampling frequency, communication delay and time-series storage; [79] and [28] for the incubator case study's use of a message broker and InfluxDB, all three in [47].
- *Protocols:* [80] weighs four common IoT messaging protocols – message queuing telemetry transport (MQTT), CoAP, advanced message queuing protocol (AMQP) and HTTP; [25] tallies which protocols predictive-maintenance twins actually use, MODBUS among them; [78] is the IoT reference architecture that places MODBUS among data-transferring capabilities.
- *The physical adapter idea:* [82] introduces the WLDT library, [83] generalises it into an edge digital twin with shadowing, [84] shows the physical adapter, shadowing function and digital adapter in an industrial case, [81] treats PT and DT lifecycles, and [49] surveys WLDT among other IoT twin platforms.
- *Agricultural twins at field scale:* [55] for an irrigation-pivot twin on an open platform, [56] for architecture patterns illustrated on agriculture and food cases, and the smart-farming section of [49].
- *Reproducible research rigs of the same family:* [31] on reproducible twin research with a small robot car, and on ocean platforms whose sensors and actuators sit on an RS-485 fieldbus behind a Raspberry Pi.
- *Consulted but not drawn on above,* and worth knowing they exist: [85] for a platform that also settled on InfluxDB for its time series, and [86] for CoAP, the constrained-device cousin of HTTP that [80] compares it with.

Chapter 3

Model, Simulation, Shadow and Twin

3.1 Before You Start

What you are assumed to know. You write software for a living, you are comfortable with HTTP APIs, JSON, a time-series or relational database, and reading Python you did not write, and you have seen a system that talks to hardware. You have read the previous two chapters, and this chapter leans on five things they established. From Chapter 1: that a **digital twin (DT)** is bought to improve one decision about one specific real system, the **physical twin (PT)**; that value comes in six patterns – *monitor, diagnose, predict, decide, control, certify* – each saying who acts on the twin's output; and that the **residual**, the gap between what a model expects and what a sensor reports, is the number most first increments are built around. From Chapter 2: the plant rig – a potted plant, a pump, and a Raspberry Pi controller that exposes everything through a REST API on port 8099 with seven endpoints that read and exactly one, `PUT /actuation/{unit}/update_schedule`, that writes – and the small mathematical model of both sides of that boundary: the pot dries at a rate d, a dose of V millilitres raises the reading by g per millilitre, and the twin predicts the next reading $\hat{m}$, compares it with the observed m_{obs}, and calls the difference r. Each of these is recalled in one clause where it is used, so a faded detail will not strand you.

What you are not assumed to know. Anything about control theory, numerical simulation, or the digital-twin literature. Where this chapter needs a term from one of those fields it defines the term at first use and then keeps using it.

What this chapter covers. The previous two chapters used the words *model*, *simulation* and *twin* freely, on one loose working definition, because their job was to say what a twin is *for* and to put a real rig on the desk. This chapter tightens the words. It separates a model from a simulation from a simulator; it gives the one test this book uses from now on to tell a **digital model** from a **digital shadow** from a **digital twin**, and shows that the test is about how data moves and about nothing else; it reads the returning arrow of that test as what it is, a control loop, and lists what closing it obliges you to build; it widens the referent from machines to software systems, processes and organisations; it sorts twins by what they twin – product, process, lifecycle, biological, ecological, sociological – and shows that the sorting is orthogonal to the test; it names five things a digital twin is not; and it classifies the plant rig at four stages of its life, ending on the single HTTP call that changes its class.

What this chapter deliberately does not cover. What is *inside* the digital object – its architecture, its components, how its model is built, solved or validated – which the chapters that follow take up. Standards in any depth: one standard supplies one definition here and nothing more. How often a twin should synchronise with its PT and how late its picture may run: both matter, both get one row in a table here, and both belong to the data chapters.

Learning objectives. By the end of this chapter you will be able to:

1. **Distinguish** a model from a simulation from a simulator, and **identify** which of the three a colleague means when they say “the model”.
2. **Classify** any proposed or existing system as a digital model, a digital shadow or a digital twin by tracing its data flows, and **justify** the classification from the direction and the automation of each flow.
3. **Apply** the classification to the hard cases – the dashboard, the offline simulator, the human who closes the loop, the referent that is not a machine – and **defend** the answer against the obvious objection.
4. **Predict** what changes in a project’s engineering obligations when it crosses from shadow to twin, and **name** the checks a command path must make before it acts.
5. **Derive** the classification of the plant rig at each stage of its life, and **specify** the one change that moves it from shadow to twin.

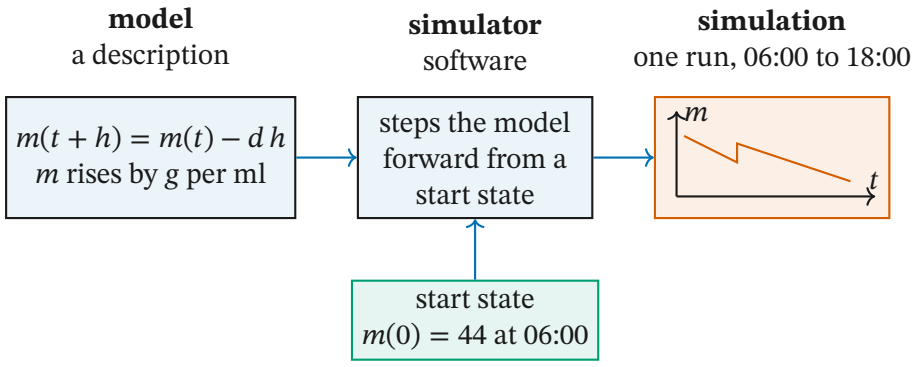
Every section that follows serves at least one of those five, and the exercises at the end say which.

3.2 Model, Simulation and Simulator

Three different things share the word “model” in ordinary speech, and in a project meeting you may be the only person who notices which one is meant. Pin them first, because the classification test in the next section depends on one of them and is defeated by confusing the other two.

> **Model.** A description of how something behaves – in equations, code, rules, a lookup table, or a fitted set of parameters. Static. It does not run.
>
> **Simulation.** One execution of a model over time, from a starting state, producing a trajectory of states. An event, not an object. You do not “have a simulation”; you *ran* one.
>
> **Simulator.** The software that performs simulations of a model.

Figure 3.1 puts the three side by side. One is a thing you can print, one is a program, and one is an event with a beginning and an end. The verbs on the bottom line are the test: if a colleague says “we have a simulation”, ask which of the other two they mean. They have a simulator, or they have a model, or they ran something once and kept the output.



You **have** a model. You **have** a simulator. You **ran** a simulation.

Figure 3.1: Three things that share one word. A model is fed, with a start state, into a simulator, and one run of it is a simulation. Only the last has a beginning and an end.

3.2.1 The rig's own instance

The previous chapter wrote down two models, and they make the distinction concrete. Both are recalled here rather than re-derived.

The **model of the physical twin** describes the pot, the pump and the controller below the API. Two of its four equations matter here. With no water, the moisture reading falls at the drying rate:

$$m(t + h) = m(t) - d\,h$$

and a delivered dose raises it in proportion to the dose response:

$$m_{\text{after}} = m_{\text{before}} + g\,V_{\text{del}}$$

Two parameters, d in percentage points per hour and g in percentage points per millilitre, and two lines. Nothing executes. That is a model.

The **model of the digital twin** describes what the software above the API believes and does. It reads a moisture m, asks for a dose V by the dose rule, and predicts the reading at the next check T_{c} hours later:

$$\hat{m} = m + g\,V - d\,T_{\text{c}}$$

then, when that reading m_{obs} arrives, computes the residual $r = \hat{m} - m_{\text{obs}}$. Also a model, and also static: three equations on a page, none of which has been run.

A note on the words, because the next section is about to reuse one of them. "The model of the digital twin" means *the twin's own equations*. "A digital model", two pages from now, will mean *a class of system* – a PT and some software with no automated link between them. The two phrases share a word and mean different things, and the chapter keeps them apart by always saying "the model **of** the digital twin" for the equations and "**a** digital model" for the class.

A **simulation** is one run of either model. Take the pot's model, start at $m(0) = 44$ at 06:00, assume the drying rate the previous chapter fitted, $d = 1.0$, and a dose response of $g = 0.1$ – a value this chapter assumes, since the previous one named g and did not fit it – and deliver one dose of $V = 100$ millilitres at 09:16. What is the reading at 18:00? Twelve hours of drying take off twelve points; the dose puts back ten:

$$m(12) = 44 - 1.0 \times 12 + 0.1 \times 100 = 42$$

That arithmetic, and the curve it traces on the way, is a simulation. Change the start reading, or move the dose to 07:00, and you have a *different simulation of the same model*. Change d and you have a different model.

The **simulator** is the dozen lines of Python that step the model forward hour by hour and hand back the curve. For a two-parameter model those dozen lines are the whole of it. Once a model has stiff dynamics, several coupled parts, or a physics library behind it, "step it forward" hides an enormous amount of machinery, and a later chapter is a survey of the kinds of simulator that exist.

3.2.2 Why the distinction earns its place

Four things go wrong when the three words collapse into one, and each has cost somebody a quarter.

People hear "twin" and picture a simulation. A survey of practitioners on standardising twins reports two-thirds of respondents agreeing that a system with no interaction running back

to the physical side should not be called a twin – and one respondent explaining the disagreement in the field by saying that, because there is no single definition, many colleagues mean simulation when they refer to digital twins [38]. A simulation is a run. A twin, as the next section defines it, is a standing pairing with one specific physical object. You can simulate an aircraft design that does not exist; you cannot twin it, because there is no particular aircraft for the data to flow from. **A twin needs a referent; a simulation does not.** That one sentence settles most arguments about whether a simulation project is "really" a twin project.

"We already have a model" is true and means less than it sounds. The modelling-and-simulation literature separates the steps that phrase hides: a conceptual model in the developer's head, an executable or programmed model that captures it, *verification* that the program implements the concept without error, and then *calibration* of its parameters against the real system [48]. A model that exists as a report has done the first step. A model that a service can call every ten minutes against live readings from *this* pot has done all four. The distance between them is where a surprising number of twin projects lose their first quarter, and the question that measures it is "can I call it, with today's readings, and get an answer back for this plant?"

A simulator is not a twin, however good it is. The engineering-of-digital-twins literature puts it bluntly: having models, and running them, gets you no twin by itself, and a model earns its place in one only when readings from the PT can be fed into it; a hardware-in-the-loop rig that never feeds its outputs back to a deployed PT is not one either, and a *tracking simulator* – one kept in step with live data – is closest to what the next section will call a shadow [45]. The standards analysts say the same thing from the other side: simulations and twins both use models to replicate a product or a process, and the most notable difference is that a twin is fed by live measurements, kept in step with the thing measured, and can talk back to the sensors that supply them, which is what separates it from conventional offline modelling and simulation [87]. Both are pointing at the same fact. The difference between a simulator and a twin is not in the model. It is in the arrows.

Models and data are complementary, not rivals. The data from the PT tells the twin what state the PT is *in*; the model tells it how the PT is *expected* to behave from that state, and the twin's picture is refreshed whenever the real thing moves on, which is what lets anyone trust conclusions drawn on it [47]. The rig's residual r is exactly that complementarity in one number: a prediction the model made, minus a reading the data supplied. A twin with data and no model can only report the present. A twin with a model and no data can only describe a generic pot. It takes both, and the next section is about the arrows that carry the data half.

3.3 Digital Model to Digital Twin

An argument about vocabulary is usually a waste of a meeting. This one is not, for two reasons that cost money. First, the term genuinely has no settled meaning: a manufacturing study of the two terms opens by admitting that nobody yet agrees what they cover, and that the disagreement gets in the way of building anything with them [1]; the practitioner survey quoted above found a two-thirds majority for "no bidirectional interaction, no twin", which is a majority and not a consensus [38]. Second, the boundary is contractual. "We will deliver a digital twin" is a sentence someone will sign. If the customer reads it as *the software will adjust the machine* and you built software that displays the machine, you have a dispute. Being able to say, at the kickoff, "what we are scoping is a digital shadow, and here is exactly what would make it a twin" turns that future argument into a present decision. This section exists so that you can say that sentence and defend it.

3.3.1 Three ingredients

Strip away every disagreement and the same three parts survive.

1. **A physical twin.** A *specific* real thing. Not a class of things – not "a potted basil" but *this* pot, on *this* desk, with the probe pushed into *this* soil. Chapter 1 pinned the term and it is unchanged. The Non-Physical Systems section widens what "physical" may mean; it never widens "specific".
2. **A digital object (DO).** The software side: models, stored state, data and services that represent that one PT. This book uses the neutral phrase *digital object* on purpose, so that classifying a system stays a question about how data moves rather than about how clever the software is.
3. **A connection.** The data flowing between the two, in one direction or both, automatically or carried by a person.

Notice what is *not* on the list: three-dimensional geometry, machine learning, a cloud platform, a user interface. Every one of those appears in real twins. None of them is constitutive. Keep the list at three, because the third item is where the whole section happens.

3.3.2 The data-flow test

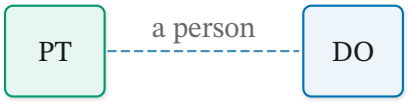
Here is the definition this book uses from now on. It comes from a categorical review of twins in manufacturing, which noticed that authors were writing *digital model*, *digital shadow* and *digital twin* interchangeably while meaning systems whose data moved in quite different ways, and gave each of the three its own name [51]. Trace the data. Ask about each direction separately, and ask whether the flow is *automated* or needs a person to carry it. In particular, look at the flow *back*: in two of the three classes it exists, and the only thing that separates them is who carries it.

> **Digital model (DM).** A physical twin and a digital object both exist, and there is no automated data flow in either direction. All exchange is manual: a change of state in the PT has no direct effect on the DO, and a change in the DO has no direct effect on the PT.
>
> **Digital shadow (DS).** The flow from the PT to the DO is automated; the flow back is not. When the PT's state changes, the DO follows on its own – but whatever travels from the DO back to the PT is carried by a person.
>
> **Digital twin (DT).** The data flows are automated in *both* directions. The PT changes the DO, and the DO changes the PT.

Pause on the shadow's return path, because it is the part most often drawn as missing. It is not missing. Somebody reads what the shadow shows and does something to the PT, or nobody would pay for the shadow; the flow from DO to PT is real, and a person is the channel. What the shadow lacks is the *automated* version of that flow, and that single word is the whole distance to a twin.

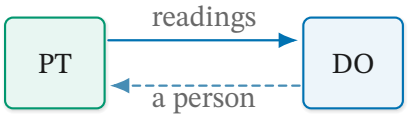
The split has been picked up widely enough that you will meet it under several phrasings, and they agree. One restatement puts it as manual exchange in both directions being a model, automatic physical-to-digital transfer with no reaction from the digital side being a shadow, and bilateral communication being a twin [88]. Another supplies the everyday instances this chapter will lean on: a computer-aided design (CAD) model of a car is a digital model, the mileage and speed on the car's dashboard are a digital shadow, and a twin needs the automated flow back [60]. An architecture-pattern catalogue frames the same three as design alternatives – data moved by

manual intervention; sensors providing the automated flow inward; the two objects connected in both directions [56] – and a reproducibility study restates them as the standard subcategories by level of integration with the PT, noting in passing that the firms it studied mostly ran the first two [31]. That much agreement across independent groups is why this is the definition worth memorising. Figure 3.2 is the whole distinction on one page: three pictures that differ only in which arrows are automated.



(a) digital model

no automated flow either way; whatever moves, somebody carries



(b) digital shadow

automated flow inward; the return exists, and a person carries it



(c) digital twin

automated flow both ways; the DO knows and acts

Figure 3.2: The data-flow test. The three classes differ in nothing except which arrows are automated – not in fidelity, not in cost, not in how impressive the visualisation is. In (b) the return arrow exists and a person carries it; the automated version of that same arrow, in (c), is the subject of the rest of this section.

Note what the figure does *not* show, because that is the point of drawing it. There is no axis for model quality, no axis for how much data is collected, and no axis for how good the dashboard looks. A crude two-parameter model with both arrows automated is a twin. A photorealistic three-dimensional replica fed by a thousand sensors, whose only way back to the PT runs through a person, is a shadow.

3.3.3 The two-question test

Reduced to something you can run in a meeting:

```
Q1. Does the physical twin's state reach the digital object
    without a person carrying it?
      no  -> DIGITAL MODEL. Stop.
      yes -> go to Q2.

Q2. Does the digital object's output reach the physical twin
    and change what it does, without a person carrying it?
      no  -> DIGITAL SHADOW. The return flow, if any, is
             carried by a person.
      yes -> DIGITAL TWIN.
```

Three properties of this test are worth naming, because each is a place people get it wrong.

It is about flows, not about sophistication. A digital object running a coupled multi-physics simulation, fed by hand from a spreadsheet once a month, is a digital model. A two-parameter drying model fed automatically from a soil probe and wired to a relay is a digital twin. The test is indifferent to how impressive the software is, which is exactly why it is useful: sophistication is what vendors compete on, and it tells you nothing about what the system will do.

"Without a person carrying it" is the load-bearing clause. The Misconceptions section works through the case where a person is in the loop, because it is the question that comes up most and the one where the honest answer is the least comfortable.

It classifies a deployment, not a product. The same codebase can be a shadow in one installation and a twin in another, if the second enables an output path the first does not. Classify what is deployed.

3.3.4 Why the line is drawn here

A student is entitled to ask why *automated bidirectionality* is the line, rather than fidelity, or real-time-ness, or having a physics model. Three reasons, in increasing order of practical weight.

It is checkable. You can answer both questions by reading a network diagram and a deployment configuration. You cannot answer "is the model faithful enough?" that way; that takes a measurement campaign and a later chapter.

It is where the risk changes. A shadow that is wrong displays something false. A twin that is wrong *does* something false. That is a different class of failure, and the next subsection is about what it obliges you to do.

It is where the cost changes. Everything in the digital-to-physical direction – authorisation, command validation, rate limiting, interlocks, rollback, an audit trail, a safety case – is work a shadow never has to do. Crossing the line is not one more feature. It is a different project.

With that, Chapter 1's loose working sentence is retired and replaced:

> **Digital twin.** A digital object paired with a specific physical twin, where data flows automatically in *both* directions – physical to digital, and digital back to physical.

Chapter 1's sentence is not deleted, it is demoted. It described what a twin is *for*. This one states what makes it a twin rather than a shadow. The book uses both, at different jobs: the earlier sentence when arguing value, this one when classifying a system.

3.3.5 The return arrow is a control loop

Look again at panel (c) of Figure 3.2. Readings go up; commands come down; the commands change the PT; the next readings reflect the change. That is a feedback loop, and a twin is one by construction. The engineering-of-digital-twins literature names the difference between a twin and a shadow in exactly these terms: a twin can support reconfiguration of the PT with some degree of autonomy, which is what lets the PT respond to changes in its environment, and the open problem is how to *close the loop* well [89]. The same loop appears in the self-adaptive-systems reading of twins: the DT monitors the physical system, updates its internal model of it, and adjusts it through controllers to hold given requirements – the twin as the managing system, the PT as the managed one [54].

Two things follow from reading the return arrow as control rather than as "output".

The first is an ordering. You build enough confidence in the twin's picture to trust conclusions drawn on it, and only then let it inform decisions that change the real world – and only where that trust has been earned [47]. Shadow first, twin second, is not timidity; it is the order in which the evidence arrives.

The second is that the return arrow carries obligations the inward arrow never did. Figure 3.3 draws the loop as the rig will run it, and puts a box on the return path that panel (c) of the earlier figure left out: the **command path**, the checks a command passes before it reaches the PT. The US National Academies' report on twins makes the same point from the definition side: the bidirectional interaction may run in real time, as for an autonomous vehicle, or slowly, as for post-flight updating of an engine twin, and the decision at the end of it may be automatic or have a human in the loop – but it is always a decision that acts [90].

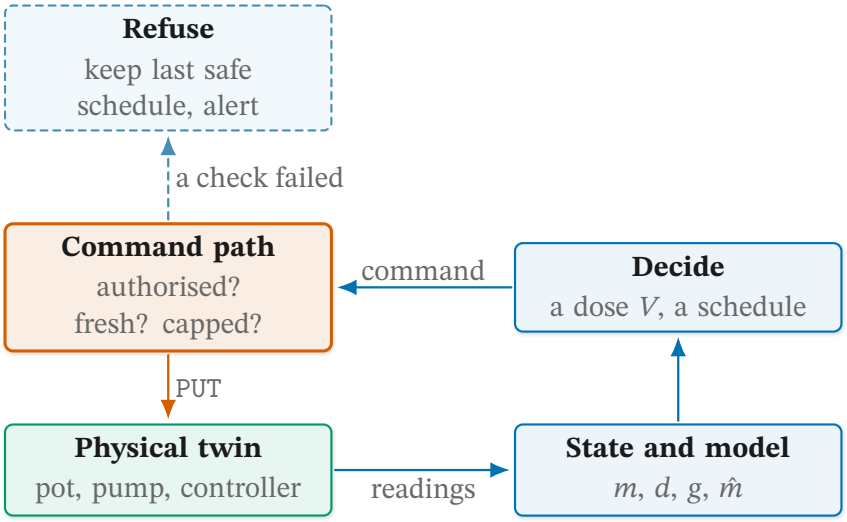


Figure 3.3: The twin as a control loop. Readings travel up the right-hand side, a decision comes back down the left, and everything in the command-path box is work a shadow never had to do. The dashed branch is what the twin does when a check fails.

Table 3.1 is the ledger of what changes. Read it as the scope difference between two quotes for the same rig.

Table 3.1: What changes when the loop closes. Every row of the right-hand column is an obligation the left-hand column never had.

	Digital shadow	Digital twin
Worst case if the model is wrong	Displays something false	Does something false
Who is accountable for an action	The person who acted	Your software
Command path	None	Authorisation, validation, rate limits, caps, interlocks
Reversibility	Not applicable	Must be designed for; some actions cannot be undone
Staleness	Shows an old number	Acts on an old number; needs a bound
Concurrency	Not applicable	Another writer may be changing the same state

	Digital shadow	Digital twin
Testing	Against recorded data	Against the PT, or a trusted stand-in for it
Failure of the connection	Stale display; degrade gracefully	Must fail safe, which you have to define
Credibility evidence	Useful	Mandatory

The row to read first is the top one, because every other row follows from it. Once the worst case is "does something false", accountability moves from a person to your software, and the command path, the staleness bound, the fail-safe behaviour and the credibility evidence all arrive at once. Four consequences for how you work:

- **The safety question becomes yours.** Before writing the command path, write down the worst thing the twin can command and what physically prevents it. Often the right answer is a limit in the controller's firmware that the twin cannot raise, rather than a check in the twin.
- **You need a state you can trust, not only data you can read.** A shadow can be sloppy about whether its state estimate is right, because a person reads the number and applies judgement. A twin cannot.
- **The failure modes are new, not merely worse.** A shadow has one interesting failure: it stops updating. A twin adds commanding on stale state, commanding on a diverged model, commanding while a person is also commanding, partial command delivery, and the loop oscillating because the twin's action changes the reading that triggered it. All of these are ordinary distributed-systems problems wearing physical consequences, which is good news for a reader with your background.
- **The incremental path is the normal path.** Model, then shadow, then twin is a sensible order, and each stage pays for the next. It is Chapter 1's lean argument in this chapter's vocabulary.

3.3.6 Other definitions you will meet

You will be handed definitions that are not this chapter's. Three are common, and the professional move is to reconcile them with the data-flow test rather than argue, because they are mostly answering different questions.

The aerospace one. NASA's, from a 2010 roadmap, is quoted across the literature in its own words: a twin is "an integrated multiphysics, multiscale, probabilistic simulation of an as-built vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its corresponding flying twin", as reproduced in [48]. It is optimised for a domain where being wrong kills people and where the twin's job is life prediction on one airframe. Note "as-built" and "its corresponding flying twin": the referent is specific, which agrees with this chapter. Read it as a description of the high-fidelity end of a spectrum, not as the entry requirement; applied as an entry requirement, almost nothing in industry qualifies, including systems that are unambiguously twins by the data-flow test and are paying for themselves.

ISO 23247's. The manufacturing standard defines a twin as a "fit-for-purpose digital representation of an observable manufacturing element with a means to enable convergence between the element and its digital representation at an appropriate rate of synchronization" [91]. Three phrases are doing work. *Fit for purpose* is Chapter 1's fidelity principle in standards language,

and the standards analysts say as much [87]. *Observable* rules out twinning what you cannot measure. *An appropriate rate of synchronisation* is the staleness row of the table above, promoted to the definition. The standard's "observable manufacturing element" is this chapter's PT under another name, and a recent survey lines the two vocabularies up side by side [60]. Take it as a refinement of the data-flow test that adds a rate question, not as a rival.

The consolidated academic one. A generalisation exercise across published definitions proposes a twin as "a virtual representation of a physical system (and its associated environment and processes) that is updated through the exchange of information between the physical and virtual systems", and shows how many different words the literature uses for each of those four parts – "counterpart", "mirror", "clone" for the representation; "asset", "process state", "vehicle" for the physical system [92]. It is the most compatible of the three and the least operational: "exchange of information" is the data-flow test's axis stated without a direction, so it will not settle an argument about a dashboard.

Table 3.2 puts the three beside the test this book uses, on the one question a project manager will actually ask.

Table 3.2: Four definitions on one question. The row that answers it is the row this book uses, and that is the only reason it was chosen over the others.

Definition	Asks	Answers "is my dashboard a twin?"
Data-flow test (this book)	Which directions are automated?	Yes – and the answer is no
Aerospace	How faithful, on a specific as-built article?	Not directly; sets a high bar
ISO 23247	Fit for purpose, and synchronised fast enough?	Partly; adds a rate question
Consolidated academic	Is information exchanged?	No; the direction is unspecified

The pattern to read off the table is in the last column: only the first row gives a yes-or-no answer that two engineers could check against a deployment diagram. The others are not wrong. They answer questions about fidelity, purpose and exchange that this book takes up later, one chapter at a time.

3.4 Non-Physical Systems

Everything so far has had a pot, a pump or an airframe at the end of the arrows. The word "physical" in *physical twin* invites that reading, and for most of this book it is the right one. But the two-question test never mentions matter. It asks whether state reaches the digital object without a person carrying it, and whether the digital object's output changes the referent without a person carrying it. Anything with a state that changes over time and can be observed and acted on can stand in the PT's place. This section walks through three families of such referents that a software engineer will meet, and shows that the test survives each move unchanged.

The literature has already made the move. A vision paper on how twins evolve prefers the phrase **actual twin** for the object, system *or process* a twin represents, precisely so that "physical" stops doing work it was never meant to do [34]. An earlier survey says the real twin may be physical or conceptual, and lists humans, urban buildings and industrial or business processes

among the referents alongside vehicles and manufactured products [52]. This chapter keeps writing PT, because the book has used it for two chapters and the rig is a real pot, and reads it broadly from here on. Figure 3.4 shows the shape the rest of this section fills in: the same loop three times, with a running service, a hospital ward and a city in the PT's slot, and the class decided each time by who carries the return arrow.

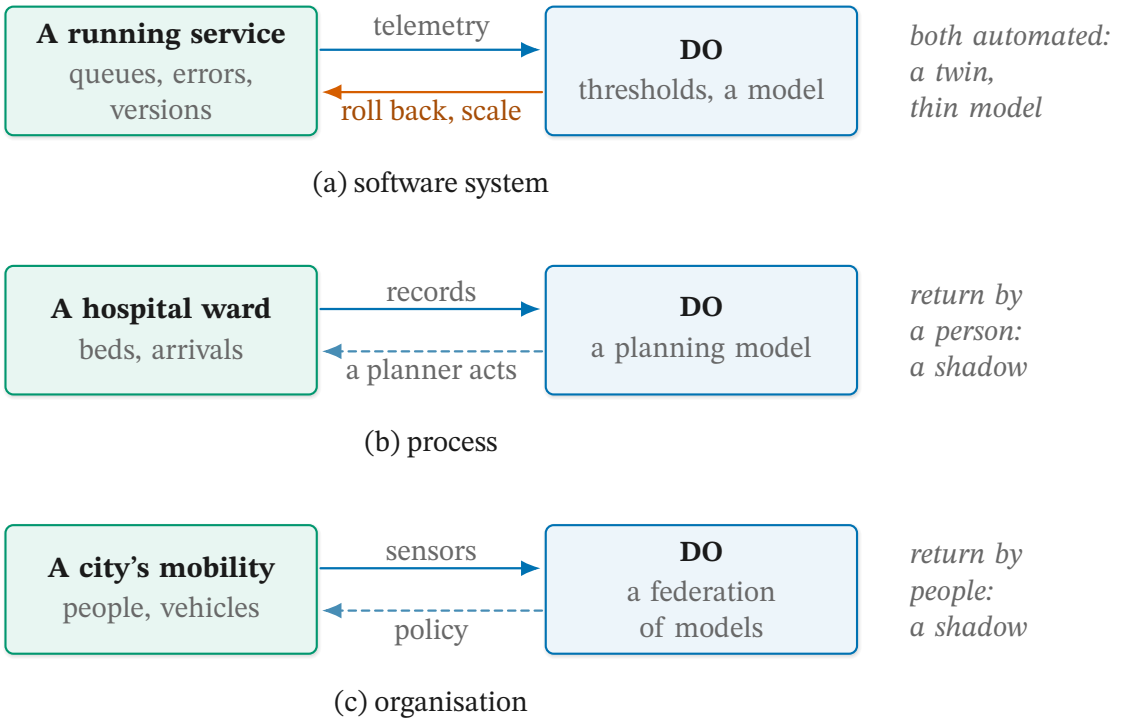


Figure 3.4: Three non-physical referents in the physical twin's slot. The inward arrow is automated in all three; only in (a) is the return arrow automated too, so only (a) is a twin.

3.4.1 Software systems

A running software system has state – queue depths, error rates, deployed versions, the configuration in force – that changes minute by minute and is observed through telemetry. It also has levers: a deployment can be rolled back, a feature flag flipped, a replica count raised. So the two questions apply. A dashboard of a service's metrics is a digital shadow of that service, by exactly the argument the Misconceptions section makes for a car's dashboard. An autoscaler that reads the same metrics and changes the replica count is, structurally, a digital twin of the service – though a very thin one, because its "model" is a threshold rather than a prediction.

Two published lines of work make this concrete. The first turns the telescope around and applies twins to the *software engineering process*. A recent position paper starts from how much of a team's DevOps activity is now recorded as data, asks what a rich enough model of that record would be, answers that it would be a twin of the software engineering process itself, and rests the answer on the same data-flow model this chapter uses: the twin would *sense* data from the process, *reason* about it with internal models, and *act* on the process, for instance by suggesting code improvements [93]. Run the test yourself. Sensing from the pipeline without a person carrying it: yes. Acting on the process without a person carrying it: if the twin opens the pull request and a human merges it, that is a shadow with a human-in-the-loop actuation path, and the paper's own vocabulary lets you say so.

The second line runs the other way and treats the twin *as* software to be operated. One group

states it as a definition – what a twin *is*, to them, is a piece of software: one that uses models and data about a cyber-physical system to represent, predict and prescribe its behaviour for a specific purpose, used at operations time as the counterpart of the design model used at development time [94]. Under the name TwinOps, others proposed unifying model-based engineering, twins and DevOps in one workflow for cyber-physical systems, on the grounds that DevOps already covers the path from development through integration to run-time monitoring and improvement [95]. The point for this chapter is the one that a software engineer will find comfortable: whatever the referent, the digital object is a service, it is built, deployed, monitored and changed like one, and its classification says nothing about how it should be shipped.

3.4.2 DevOps for a building's twin

The DevOps thread reaches physical referents too, and a building is the instructive case because a building is physical, large and slow, and its twin is neither. A framework for the engineering of twins of *built assets* – a multi-functional room, a climatic chamber – starts from the observation that a twin is complex software that must be continuously modified to meet evolving requirements, and supplies the continuous-integration and continuous-delivery infrastructure to do so [96]. A campus twin from another group brings a university's buildings, their day-to-day running and what-if simulations under one roof, and makes a point this chapter needs: the fixed description of a building has long lived in its information model, while the live feeds from its sensors and its automated controls have had nowhere to live and nobody acting on them [97]. Run the test on that campus twin: sensors deliver state automatically, so Q1 is yes; whether any setpoint is written back without a person decides Q2, and the paper describes a preliminary platform focused on energy use, which reads as a shadow. Two things to carry away. A building's twin is a software product with a release cadence measured in weeks, wrapped around a referent whose own changes are measured in decades. And DevOps applied to the twin's software is a different thing from a twin of a DevOps process: the same practice, pointed at different referents, and the test tells them apart.

3.4.3 Processes, organisations, cities

The widest move is to a referent with no fixed hardware at all. A hospital's patient flow is a process: patients arrive, need beds, are discharged, and the ward's state is a set of occupied and free beds that changes hourly. A twin built at the University of Oslo turns that stream of arriving patients into a stream of optimisation problems, solved by a constraint solver over an executable formal model, to plan ward needs and explore what-if scenarios [98]. A related effort mirrors a hospital's operating suite – rooms, resources, staff – as an ecosystem of twins integrated with the hospital's legacy systems through a healthcare data standard [99]. Neither has a pump. Both have a referent with observable, changing state, both read that state automatically, and both hand their output to a planner, which by the test makes both shadows with a human-in-the-loop actuation path. That is not a criticism; it is the correct class for a system whose actuator is a hospital administrator.

The same holds one level up. A supply chain is a network of organisations, and twins of supply chains let managers build alternate realities of their network and explore its vulnerabilities, a digital form of wargaming for resilience [2]. A city's mobility is travellers, their vehicles and the roads and rails under them, all at once, and one federation of twins built to weigh up interventions in it was shaped by a plain fact: automating such decisions conflicts with how planning is actually done – so it keeps human-in-the-loop control throughout and writes down which steps are automated and which are human decisions [100]. That last design choice is the Misconceptions section's argument arriving from the other direction: when the referent is a society, the person in the loop is a requirement, not a gap.

3.4.4 What the move costs

Widening the referent costs nothing at the level of the test and a great deal at the level of the model. A pot's state is one number from one probe. A hospital ward's state is hundreds of records in a legacy system, some of them wrong. A city's is contested. The National Academies' definition of a twin explicitly covers "the engineered world, natural phenomena, biological entities, and social systems", and in the same breath insists on *predictive capability*: a twin "must be able to issue predictions beyond the available data to drive decisions" [90]. The less the referent resembles a machine, the harder such a prediction becomes to make, and to believe, and the more often a working system's honest class turns out to be *shadow*. Keep the two facts separate. The test does not care what the referent is made of. The model does.

3.5 Types of Digital Twins

Once the referent can be anything with observable state, it helps to have names for the kinds of referent people actually twin. The literature has several such schemes. One review catalogues two: a shop-floor scheme of *product*, *process* and *operation* twins, and a more general one of *object*, *process* and *phenomena* twins [101]. Another review's definition admits three kinds of referent at once: something made, something offered or something done – product, service, process [21]. This section uses six labels that cover what a software engineer will meet – product, process, lifecycle, biological, ecological, sociological – and makes one point about all of them before it starts.

Every one of these is orthogonal to the data-flow test. A type names *what* is twinned. The test names *how data moves*. A product twin can be a model, a shadow or a twin; so can a process twin; so can a twin of a lagoon. The professional sentence states both: "a process twin of the ward, currently operating as a shadow." That sentence is precise, and it is the kind of sentence that ends a circular meeting. Figure 3.5 draws the relationship: six kinds of referent on the spokes, one test at the hub.

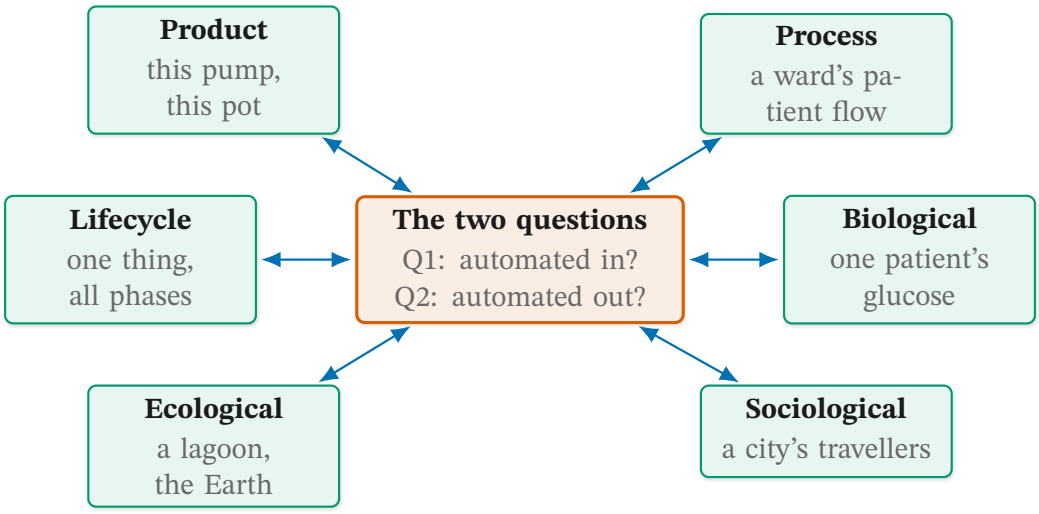


Figure 3.5: Six kinds of referent, one test. Whatever sits on a spoke, the two questions at the hub decide whether it is a model, a shadow or a twin.

Product twins. The referent is one manufactured thing: this pump, this turbine, this airframe. The rig's plant twin is a product twin in this sense – the referent is one pot – and so is every

twin the aerospace definition had in mind. The observable state is the thing's condition; the return arrow, if there is one, changes how the thing is operated or maintained. Most of Chapter 1's deployment families were product twins.

Process twins. The referent is an activity with a beginning and an end that repeats: a production line's cycle, a fermentation, a surgery's turnaround, a patient's stay. The standards analysts note that a twin can be created for processes as well as products, with simulations run on real-time process data [87], and the hospital twins of the previous section are process twins. Their state is a schedule or a queue rather than a temperature, and their actuator is usually a person, which is why they classify as shadows so often.

Lifecycle twins. The referent is one thing followed *across* the phases of its life rather than in one of them. A review of twins along the product lifecycle splits that life into design, production, use and disposal, and finds that most published twins live in the middle two phases, as prototypes focused on real-time observability, with the design and disposal phases still thin [102]. Two complications make lifecycle twins harder than they sound. The referent itself has phases – a machine in commissioning sends telemetry at a different rate and range than one in service, and a twin that does not model those phases may read an expected change as a failure [81]. And the twin has a life of its own that must be kept in step with the referent's; the engineering literature treats that continuous, mutual evolution as the core problem of building twins at all [34]. The same idea reaches software: a twin could accompany a software system through its own life cycle, and with a product line there could be one twin for the reference design and one for each product built from it [103].

Biological twins. The referent is an organism or part of one, most often a patient. The claims here run ahead of the deployments, and a careful reader should hold the two apart. On the deployment side, a roadmap for immune-system twins points to twins of the human heart in diagnosis and to the artificial pancreas, which manages insulin for Type I diabetic patients, as operational medical twins [104]. Run the test on the artificial pancreas: a glucose sensor delivers state automatically, a model decides a dose, a pump delivers it. Both arrows automated, one specific patient. It is a digital twin by this chapter's definition, and it is one of the few in medicine that is. On the aspiration side, a European position paper sets out a *Virtual Human Twin* infrastructure for developing and validating healthcare twins and for personal health forecasting [44], and a perspective on personalised medicine argues that data-driven methods alone cannot deliver it and must be complemented by model-driven patient twins [105]. The data caveat is severe: twins that track a person's health over time need longitudinal data, which is scarce, fragmented across institutions and noisy [14]. Most of what is called a patient twin today is, by the test, a digital model or a shadow, and the perspective papers say as much between the lines.

Ecological twins. The referent is an ecosystem or the planet. A marine twin of a Mediterranean coastal lagoon models the lagoon's ecosystem, uses learned models to emulate its hydrology, and runs hypothetical scenarios for managers, motivated by human and climate pressures on the coast [106]. At the largest scale, the European Commission's Destination Earth programme is building a high-precision twin of the Earth system for climate adaptation and disaster-risk management, one of several scientific twin initiatives that have followed the concept out of industry into Earth-system science and biomedicine [107]. A collection of commentaries on Earth twins adds the sobering condition: to be more than big-data atlases, such twins need a new Earth-system simulation and observation capability, and they must model the Earth *and* the humans acting on it together [108]. Apply the test and the answer is instructive. Observations reach the digital object automatically, so Q1 is yes. Nothing the digital object outputs changes the ocean without a person, so Q2 is no. A twin of Earth, in this chapter's vocabulary, is a very large digital shadow whose only actuator is policy – and its designers know it, which is why the human-interaction problem is the one they name.

Sociological twins. The referent is people in the aggregate: a city's travellers, a supply

network's firms, a hospital's staff. The urban-mobility federation of the previous section is the clean example, and its defining design decision – human-in-the-loop control throughout, with the automated and human steps written down [100] – is the right one for a referent that can read the twin's output and change its behaviour in response. The mobility twin from another group goes one step further and gives the human a twin of their own, a *Human Digital Twin* that classifies driver type and feeds a personalised cruise controller, alongside twins of the vehicle and the traffic [109]. When the human is the referent, the return arrow is a recommendation, and a recommendation a person may ignore is, by the test, a person carrying the flow.

Table 3.3 collects the six, with the one question to ask of each.

Table 3.3: Six kinds of referent. The last column is what the test usually returns for published systems of that type; none of the types forces a class.

Type	Referent	What the inward arrow carries	Usual actuator	Class most often met
Product	one manufactured thing	condition, load, position	a controller or a maintainer	shadow or twin
Process	one repeating activity	schedule, queue, cycle time	a planner	shadow, human in the loop
Lifecycle	one thing across its phases	phase, configuration, history	varies by phase	model early, shadow later
Biological	one organism or organ	physiological signals	a clinician; rarely a device	model or shadow; the artificial pancreas is a twin
Ecological	an ecosystem, the planet	observations, satellite data	policy	shadow
Sociological	people in the aggregate	movements, transactions	the people themselves	shadow, human in the loop

Read down the last column and one pattern stands out: the further the referent is from a machine with an actuator, the more often the honest class is *shadow*. That is not a failure of the field. It is the test doing its job, and telling you where a person still has to stand.

3.6 Misconceptions on Digital Twins

Negative definitions teach faster than positive ones, because the reader arrives with the wrong ones already installed. Five things a digital twin is not, each with the tell that exposes it.

3.6.1 Not a 3D model

Geometry is a representation of *shape*. A twin is about *state and behaviour*. The two get conflated because twin marketing is visual, and a rendered plant floor photographs better than a residual plot. Visualisation is genuinely valuable: a good picture shows a person a shape in the data that

a table of numbers hides – a drift, a repeating pattern, two channels moving together – and with it the trouble that shape may be announcing; the useful forms run from plain charts through animated schematics to three-dimensional scenes [110]. Note the verb: it *presents*. Presentation is a service the twin offers, and a later chapter covers it properly. It is not the twin.

The tell: ask what happens if you delete the geometry. If the twin still answers its questions, the geometry was a view. If the whole thing stops, you had a Computer-Aided Design (CAD) model with a data feed – which the taxonomy's own examples file under *digital model* when the feed is manual and *shadow* when it is not [60].

3.6.2 Not a monitoring dashboard

Chapter 1 said, of the *monitor* pattern, that many systems sold as twins stop there. Settling whether that counts:

A monitoring dashboard is a digital shadow, not a digital twin.

Run the test. Q1: does the PT's state reach the digital object automatically? Yes; that is what a dashboard is. Q2: does the digital object's output change what the PT does, without a person carrying it? No. A dashboard's output goes to a human retina. Digital shadow. The taxonomy's own illustrative shadow is exactly this – the mileage and speed on a modern car's dashboard [60].

Three things follow, and the third is the one that matters commercially. *This is not an insult.* Chapter 1's worked example, the dosing-rig fault detector with a nine-month payback, is a shadow. Shadows earn money, and reviews of the field consistently find more published work on models and shadows than on twins proper [51]. One review went further and measured it: over half of the papers it examined described a model or a shadow while claiming a twin, so it distinguishes the *claimed* twin, the *described* twin and the *actual* twin, and finds that most claimed twins are, in their case studies, models or shadows [101]. *But you must not sell a shadow as the other thing.* The category difference is real, the customer's expectation attaches to the word, and the gap is exactly the right-hand column of the obligations table. *And the honest sentence is short.* "This is a digital shadow. It will tell you within a check interval that a dose did not land. It will not reschedule the dose. Making it do so is a separate increment, and here is what that costs."

3.6.3 Not a twin because a human acts on it

The hard case. A shadow detects the fault, raises an alert, a technician reads it and edits the watering schedule. The loop *is* closed – through a person. Twin or shadow?

By the data-flow test, shadow. The test says "without a person carrying it", and a person is carrying it – which is not a loophole but the definition: a shadow's return path is the one a person carries.

That answer feels legalistic, so here is why the line is drawn where it is, which is the part worth learning. When a human is in the loop, the human applies judgement the digital object does not have and rejects commands that are plainly wrong; the system's response time is bounded by human availability – shifts, weekends, holidays – and that bound is usually the thing being bought; and accountability rests with an identified person. Remove the human, and all three move into your software. Those three differences are not vocabulary. They are the difference between two systems with different failure modes, different response times and different liability, and the test tracks that difference under a word that sounds pedantic.

Be precise rather than dogmatic. A survey of twins used for AI simulation sorts what it found into three styles – *twin*, *shadow* and *human in the loop* – and reports that the autonomous style dominates, with a minority where a person supervises or acts [111]; a mobility twin has been

demonstrated with the driver deliberately kept in the loop and given a twin of their own [109]; and one of the taxonomy's own restatements observes that unless a twin is fully autonomous, humans will need to interact with it [88]. The professional formulation is therefore not "that is only a shadow" but **"a digital shadow with a human-in-the-loop actuation path"** – which says exactly what exists, credits what it delivers, and makes the remaining gap to a twin visible.

3.6.4 Not a simulator

The first section covered the vocabulary; here is the operational tell. A simulator answers questions about a *class* of systems or a hypothetical one. A twin answers questions about *this* system, the one on the desk, in the state it is in now. A simulator with no referent and no live data is a digital model in this chapter's sense, however good it is. The engineering literature's warning bears repeating – a model plus a run of it is still not a twin [45] – and the standards analysts agree on where the line falls: in the live measurements and the path back to the sensors, not in the model [87].

A crisp diagnostic: **ask which physical object would have to be destroyed for the software to become meaningless.** If there is no such object, you do not have a twin. This is not a criticism of the simulator, which is usually the most valuable asset the project has and the thing a twin gets *built around*.

3.6.5 Not a data lake, and not a machine-learning model

Storage is not a twin: a historian holding five years of readings has no model, so it cannot answer a question about a state that was never measured. A trained network is not a twin either: it is a *model*, one candidate for the model slot inside a twin, and a later chapter covers what changes when a learned model sits there instead of the pot's two equations.

The general form of all five: each names a *component* or a *service* of a twin and mistakes it for the whole. Geometry, the dashboard, the simulator, the store and the learned model are all things twins contain. None of them is what makes the thing a twin. The connection is.

3.7 Classification of the Plant Demonstrator

Chapter 2 put the rig on the desk and wrote down its model. This section classifies it – at four stages of its life, with the reasoning at each step, using nothing but the two-question test and the rig's own API. It is the chapter's fully worked example; the Faded Examples section then takes the scaffolding away.

Two facts from the previous chapter carry the argument. The PT is the pot, plant, soil, pump, tubing and the Raspberry Pi controller. The controller exposes a REST API on port 8099 whose endpoints include:

```
GET /sensing/{unit}/{parameter}   readings; limit, since_timestamp
GET /actuation/{unit}/watering_events  doses delivered
GET /actuation/{unit}/show_schedule    the schedule now in force
PUT /actuation/{unit}/update_schedule  replace the watering schedule
```

Hold that list in view. The classification boundary of this entire chapter runs between the third line and the fourth.

The pot-and-pump setting is a recognised exemplar for exactly this kind of reasoning. A published greenhouse twin has, on its physical side, potted plants with probes in the soil and pumps to water them, and on its digital side a loop that both steers the pumps and keeps the

twin's own models honest [53]; the same group's later work on the same rig reads the twin and its physical counterpart together as a self-adaptive system in which the twin monitors, updates its model, and adjusts the physical system through controllers [54]. The rig in this book is a smaller member of that family. Nothing below is a toy argument about a toy.

3.7.1 Stage 0: today, as shipped

What exists. The controller samples its sensors, writes readings to the time-series database on the Raspberry Pi, and fires the pump on a fixed daily schedule. Nothing else.

Classify it. The trap here is to call the controller's database a digital object and start answering Q1. It is not one. Recall the three ingredients: the PT is *the pot, plant, soil, pump, tubing and the controller*. The controller's own logging is the PT keeping records about itself, the way a car's odometer is not a twin of the car.

Verdict: no digital object at all. Not a model, not a shadow, not a twin. This matters more than it sounds. A project that starts here and believes it is starting at "shadow" underestimates itself by an entire increment of data-engineering work.

3.7.2 Stage 1: a model in a notebook

What exists. You export a few weeks of readings, fit the two parameters d and g of the pot's model in a notebook, and produce a chart of predicted against measured moisture. You do this by hand, once. The simulation worked in the first section – $m(0) = 44$ at 06:00, one dose, a predicted $m(12) = 42$ at 18:00 – is exactly what this notebook draws, against the reading the probe actually gave.

Run the test. Q1: does the PT's state reach the digital object without a person carrying it? **No.** You carried it, by exporting a CSV.

Verdict: digital model. A useful one. It is where you discover whether a two-parameter model can represent this pot at all, which is the cheapest thing to find out early and the most expensive to discover at Stage 2. Note that this stage is neither free nor skippable, and that its output is knowledge rather than software.

3.7.3 Stage 2: the fault detector

What exists. A service that every T_c hours calls `GET /sensing/plant_1/moisture` and `GET /actuation/plant_1/watering_events`, steps the model forward, computes the residual $r = \hat{m} - m_{\text{obs}}$, and raises an alert when a scheduled dose produces no rise in the reading. This is the previous chapter's *diagnose* increment: the reading that arrives after a dose is compared with the reading the model expected, and a residual that stays positive says less water is arriving than was asked for.

Run the test. Q1: state reaches the digital object automatically? **Yes.** The polling loop, over HTTP, with no person in it. Q2: does the digital object's output change what the PT does, automatically? **No.** The output is an alert to a technician, who reads it and, if the pot is dry, edits the schedule by hand. The return flow exists; a person carries it.

Verdict: digital shadow.

Sit with that for a moment, because it is the most important sentence in this chapter:

The payback-positive first increment is not a digital twin.

That is not an embarrassment and not a bait-and-switch. It is the normal, correct shape of a first increment, and it is why the Misconceptions section insisted that calling something a shadow is not an insult. It also means that if the project was sold as "we will build you a digital

twin", Stage 2 does not deliver what was promised – while delivering exactly what was *valued*. Catching that mismatch at the kickoff rather than at the acceptance review is the practical payoff of this whole chapter.

The technician's edit is the loop closing through a person, and it is what makes Stage 2 a shadow rather than a twin. The Misconceptions section gives you the precise phrase for it, *a digital shadow with a human-in-the-loop actuation path*.

3.7.4 Stage 3: closing the loop

What changes. One thing. The service stops sending an alert and starts calling

```
PUT /actuation/plant_1/update_schedule
```

with a schedule it computed itself – the target-moisture rule of the previous chapter, $V = \min\big(V_{\max},\ k\,(m_{\mathrm{t}} - m)\big)$, wrapped as a one-event schedule.

Run the test. Q1: yes, unchanged. Q2: **yes.** The digital object's output now reaches the PT and changes what it does, with no person in the path.

Verdict: digital twin. The whole chapter, in one HTTP verb.

That is deliberately anticlimactic, and the anticlimax is the lesson: the categorical difference is a small change in the *data path* and a large change in *everything around it*. Walk the obligations table against this one call and the size of the real change appears.

- **Worst case.** The model diverges, the twin concludes the pot is dry, and it writes a schedule with a large dose every hour. What physically stops it? Today, nothing but the water tank's volume. The cap $V_{\max}$ in the twin helps; a hard cap on total daily dose belongs in the *controller*, because the twin is the component whose correctness is in question.
- **Authorisation.** The endpoint accepts a `PUT` from anything that can reach port 8099. That is acceptable for a shadow that never calls it. It is not acceptable once a program calls it unattended.
- **Staleness.** The `since_timestamp` parameter on the sensing endpoints lets the twin know how old its state is. A twin must refuse to command on state older than some bound, and the bound follows from the decision – doses are hours apart, so a bound well inside one check interval – not from the phrase "real time".
- **Concurrency.** `update_schedule` *replaces* the schedule. If a technician edits it while the twin also writes, the last writer wins, silently. In a shadow this cannot happen. In a twin it is a data race with a plant at the end of it.
- **Reversibility.** A schedule can be rewritten. Water already pumped cannot be unpumped. Design the command path around the irreversible half.

Notice that none of those five is a modelling problem. All five are distributed-systems problems – authorisation, staleness, concurrency, idempotency, blast radius – which is why the later, architectural part of this book is the deep part for a reader with your background. The command-path box of Figure 3.3 is where all five live.

3.7.5 The classification table

Table 3.4 collects the four stages, and Figure 3.6 draws them, so the shape of the climb is visible: each stage adds exactly one arrow, and the last stage adds none – it automates the one a person was carrying, and that alone changes the class to *twin*.

Table 3.4: The rig at four stages. Between stages 2 and 3 neither the model nor the existence of a return flow changes; only who carries that flow does.

Stage	What exists	Q1: automated in?	Q2: automated out?	Class
0	Controller only, fixed schedule	no digital object	–	none
1	Hand-fitted model in a notebook	no	no	Digital model
2	Polling fault detector; a technician acts on its alert	yes	via a person	Digital shadow
3	Detector writes the schedule with `PUT`	yes	yes	Digital twin

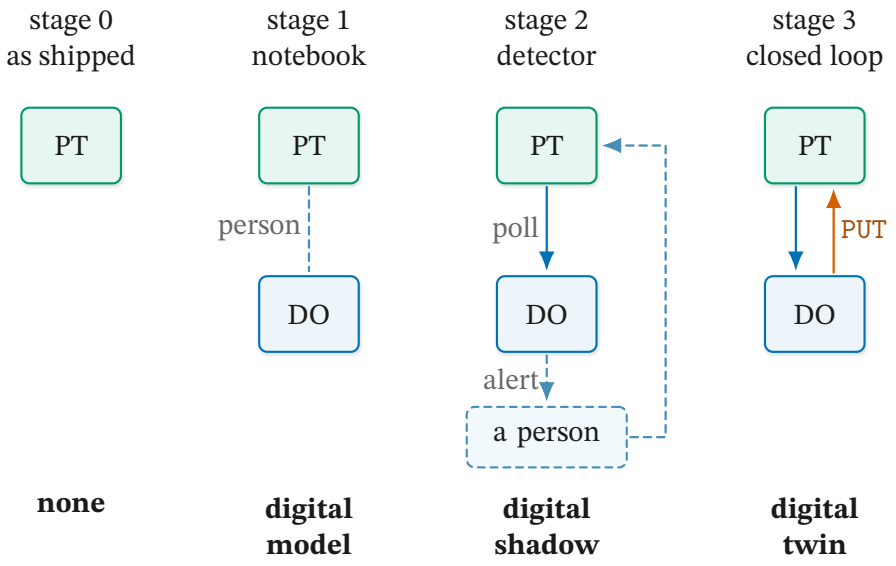


Figure 3.6: The rig's climb. One arrow per stage; in stage 2 the return arrow already exists and a person carries it. The jump that matters is from stage 2 to stage 3, where that arrow becomes automated, and it is a jump in obligation, not in modelling effort.

Read the table's second column against its last. The model at Stage 3 is the same two-parameter drying model as at Stage 1. What changed is who is allowed to act on it, which is why the obligations table is the expensive part of the picture rather than the box marked DO.

One last observation, and it is the rig's own doing. Its documentation names two kinds of user, and describes the second, the *remote connecter*, as either a person or a computer running a digital twin. The PT was designed with the Stage 3 seam already cut. Recognising that seam in somebody else's system, before you write any code, is the skill this chapter has been teaching.

3.8 Faded Examples

Same method, less scaffolding. For each system, name the PT, the digital object, answer Q1 and Q2, and classify. The first is worked in full, the second is started and left for you to finish, and the third is posed as a problem.

3.8.1 Example 1, worked in full: the hotel energy report

A hotel chain's engineering team receives a monthly PDF comparing each hotel's energy use against a building-physics model of that specific building. An analyst exports meter data, runs the model, and writes the report.

PT: one specific hotel building. *Digital object:* the building-physics model plus the exported data. *Q1:* does meter data reach the model automatically? No; an analyst exports it. Stop. **Digital model.** Note that the model is sophisticated and building-specific, and that changes nothing: it is the first property of the two-question test, flows and not sophistication. Note also what would change the answer. Wire the meters to the model and Q1 flips to yes: a shadow, with no change to the model at all. Let the model write setpoints to the building's heating controller and Q2 flips too, and the obligations table arrives with it.

3.8.2 Example 2, one step left to you: the offshore turbine anomaly service

Vibration and temperature stream from an offshore wind turbine to shore continuously. A service compares them against a model of that turbine and raises a work order in the maintenance system when the residual exceeds a threshold. A planner schedules a vessel.

PT: turbine number 12. *Digital object:* the model, the streaming state and the anomaly service. *Q1:* **yes**; the stream is automatic.

Q2 is yours. Careful: the work order *is* automatic, and it does reach a physical outcome eventually. Does it change what the turbine does without a person carrying it?

- **(a)** Classify the service and justify it in one sentence.
- **(b)** The vendor calls it a digital twin. Write the two-sentence reply you would give the customer that is accurate without being combative.
- **(c)** Name the single change that would make the service a twin under the test, and one thing that change would oblige the vendor to build that they do not need today.

Hint for (b): the Misconceptions section supplies a phrase that describes what exists without conceding the label.

3.8.3 Example 3, posed as a problem: the datacentre cooling controller

Rack inlet temperatures stream to a service every second. The service runs a thermal model, computes a set of fan speeds and chilled-water setpoints, and writes them to the building management system. Operators can override at any time but rarely do. Once a week the service retrains its thermal model on the last week of data.

Classify it, unaided. Then answer two harder questions. Does the weekly retraining change the classification? And if the operators exercise their override twice a year, does the system stop being a twin on those two days?

3.9 Summary

Back to the objectives this chapter opened with.

1. **Model, simulation, simulator are three things.** A description; one execution of it from a start state; the software that executes it. The rig's two equations are a model, the run from $m(0) = 44$ to $m(12) = 42$ is a simulation, and the dozen lines that step it are a simulator. The sentence that settles most arguments: a twin needs a referent, a simulation does not.
2. **Classification is a question about data flows.** No automated flow in either direction: digital model. Automated physical-to-digital, with the return carried by a person: digital shadow. Automated in both directions: digital twin [51]. Two questions, answerable from a deployment diagram, indifferent to how sophisticated the software is – and that indifference is the feature.
3. **The hard cases have answers.** A dashboard is a shadow, and that is not an insult. A human closing the loop leaves it a shadow, best described as *a digital shadow with a human-in-the-loop actuation path*. A simulator with no referent is a model. A referent need not be a machine – a software process, a hospital ward, a city, a lagoon – and the test is unchanged; only the model gets harder. Twins sort by referent into product, process, lifecycle, biological, ecological and sociological types, and every type can be a model, a shadow or a twin.
4. **Crossing the boundary changes the project, not the feature list.** The return arrow is a control loop. A wrong shadow displays something false; a wrong twin does something false. Authorisation, a staleness bound, concurrency, reversibility, fail-safe behaviour and credibility evidence all arrive at once, in the command path, and trust is earned on the shadow before the twin is allowed to act.
5. **The rig classifies cleanly, and the result is uncomfortable in a useful way.** Stage 0: no digital object. Stage 1: digital model. Stage 2, the payback-positive fault detector: digital *shadow*. Stage 3: one `PUT /actuation/{unit}/update_schedule` makes it a digital twin, and that single call drags five distributed-systems obligations in behind it.

The sentence to carry forward, and to say out loud at your next kickoff: **"What we are scoping is a digital shadow, and here is exactly what would make it a twin."**

3.10 Exercises

Solutions or hints follow the exercises; each says which. The objective each exercise serves is given in brackets.

Exercise 1 (Objective 2, easy). Classify each, in one line, with the flow that decides it: (a) a car's dashboard showing speed and mileage; (b) a finite-element model of a bridge, run once during design; (c) a thermostat that reads a room sensor and drives a boiler; (d) a weather forecast.

Exercise 2 (Objective 1, easy). A colleague says "we already have the model of the pot, so the twin is mostly done." Name three distinct things that sentence might mean, and the one question you would ask to find out which.

Exercise 3 (Objective 2, easy). The six value patterns are monitor, diagnose, predict, decide, control and certify. For each, state the *minimum* class from this chapter that can deliver it, and say which pattern is the first that requires a twin rather than a shadow.

Exercise 4 (Objective 3, medium). Complete Example 2 of the Faded Examples section, parts (a) to (c).

Exercise 5 (Objective 3, medium). Do Example 3 of the Faded Examples section, including both harder questions.

Exercise 6 (Objective 3, medium). A continuous-integration server runs a team's test suite on every commit, and a bot posts a comment on the pull request when a test that passed on the main branch fails. Classify it. Then: the same bot is given permission to revert the commit

automatically. What class is it now, and which three rows of the obligations table have now become the team's problem?

Exercise 7 (Objective 4, medium). For the rig at Stage 3, write the staleness rule: the maximum age of the moisture reading at which the twin is still allowed to write a schedule, and the reasoning that justifies it from the check interval T_c. State what the twin should do when the reading is older than that.

Exercise 8 (Objective 4, medium). The Stage 3 discussion lists a data race: `update_schedule` replaces the schedule, so a technician and the twin can overwrite each other. Propose two fixes – one purely in the twin, one requiring a change to the controller's API – and say which you would advocate and why.

Exercise 9 (Objective 3, medium). Pick one of the six types in the Types of Digital Twins section and construct, in three sentences each, a system of that type that is a digital model, one that is a digital shadow, and one that is a digital twin. The three should share a referent.

Exercise 10 (Objectives 2 and 4, hard). Take a system you have personally built or operated that talks to hardware, or that acts on a running software system. Classify it with the two-question test. Then: if it is a shadow, what exactly would closing the loop oblige you to add? If it is a twin, which rows of the obligations table did the project actually discharge, and which did it skip?

Exercise 11 (Objective 3, hard). The Digital Model to Digital Twin section claims the three outside definitions "mostly answer different questions". Argue the opposite: construct a concrete system that one of the three would call a digital twin and the data-flow test would not, and say which definition you would use in a contract and why.

Exercise 12 (Objectives 2 and 4, hard, open-ended). Your company is about to sign a statement of work whose draft says: "The Supplier shall deliver a Digital Twin of the Customer's bottling line, providing real-time visibility and enabling optimisation of line performance." Rewrite that sentence as three numbered deliverables that are individually testable, using this chapter's vocabulary. At least one must be a digital shadow, at least one must state a synchronisation requirement derived from a decision rather than from the phrase "real-time", and at least one must make explicit whether anything is written back to the line. Then write the two sentences you would say when the customer asks why you replaced their sentence with three of yours.

3.10.1 Solutions and hints

Exercise 1 (solution). (a) Digital shadow: automatic in, nothing automatic out; this is the taxonomy's own worked example [60]. (b) Digital model: no automated flow, and note that it is also the most technically demanding item on the list, which is the first property of the test again. (c) Digital twin, formally: sensor in, boiler command out, both automatic, one specific room. If that feels wrong, the discomfort is informative. The test is about structure, not scale, and a thermostat has the structure. What it is missing is any model beyond a setpoint, and how good the model is belongs to the *fidelity* question, not to classification. (d) Neither: there is no specific PT. The planet is not a serial number, and a forecast is a simulation. A twin of one specific building, or of one specific lagoon, is a different matter.

Exercise 2 (hint). The first section. The three meanings are roughly: the two equations on paper (a description); code that steps them forward on demand (an executable model, with a simulator); and a *calibrated* executable model whose d and g have been fitted to this specific pot. The question to ask: "can I call it with today's readings and get an answer back for *this* plant?" The workflow those three collapse – conceptual model, executable model, verification, calibration – is the one the modelling-and-simulation literature lays out [48].

Exercise 3 (solution). Monitor: shadow. Diagnose: shadow. Predict: shadow; a forecast

is still output to a person. Decide: shadow if it advises, twin if it acts, so the answer depends on the deployment, not the pattern. Control: twin, by definition of the pattern. Certify: often neither, since the PT may not exist yet, and a simulator with no referent is a digital model. The first pattern that *requires* a twin is *control*. Worth noticing: four of the six patterns are deliverable by a shadow, and the fifth sometimes is.

Exercise 4 (solution, then hints). (a) **Digital shadow.** The work order is automatic, but it reaches a *planner*, not the turbine; nothing changes what the turbine does without a person. (b) *Hint:* use the construction "a digital shadow with a human-in-the-loop actuation path", then say what it does deliver, which is genuine and valuable. Do not open by disputing the label. (c) The change: the service commands the turbine directly, for example derating or stopping it. New obligations include, at minimum, an authorisation path, a staleness bound and a fail-safe behaviour when the link to shore drops – none of which a work-order-raising service needs, because a planner supplies all three implicitly.

Exercise 5 (partial solution). Digital twin: setpoints are written automatically to the building management system. On the harder questions: (i) weekly retraining does not change the classification, because the test is about data flows between physical and digital, not about how the model was obtained; it does change the credibility problem substantially, since a model that changes weekly must be re-validated weekly. (ii) No. The classification describes the system's designed data paths, not each day's events. An override is the human-authority mechanism the obligations table says a twin needs; exercising it is the system working as designed, not the system changing category.

Exercise 6 (solution). With the comment only: a digital shadow of the codebase-under-test – state (test results) reaches the bot automatically, the bot's output reaches a person. With automatic revert: a digital twin, structurally; the bot's output changes the referent without a person. The three rows: *worst case* (a flaky test now reverts a good commit), *concurrency* (the bot and a developer pushing to the same branch), and *reversibility* (a revert of a revert is not a no-op once other commits have landed on top). Note that this is Exercise 1(c)'s thermostat in software: the structure of a twin, a model no richer than a threshold.

Exercise 7 (hint). Doses are hours apart and the quantity the twin acts on changes on a scale of hours, so a defensible bound sits well inside one check interval – an hour or two when $T_c = 6$ – and is justified from the interval, not from a preference. The second half is the more important half: on exceeding the bound, the twin must *not* write, must leave the last known good schedule in force, and must raise an alert. "Fail to the last safe state and tell someone" is the general form, and it is the dashed branch of the control-loop figure.

Exercise 8 (hint). In the twin: read `show_schedule` immediately before writing and abort if it is not what the twin last wrote – a compare-and-swap done badly, since it still races between the read and the write. In the API: add a version or an entity tag to the schedule and have the controller reject a `PUT` that carries a stale one. Advocate the API change: the race is in the resource, and a fix that lives only in one client stops working the moment there are two. Put the invariant where the state is.

Exercise 9 (hint). The easiest type is *product*. The same pump: its datasheet curve in a spreadsheet (model); a vibration feed into an anomaly service that emails a technician (shadow); the same service writing a speed limit to the pump's drive (twin). Check that your three share one referent and differ only in the arrows.

Exercise 10 (open). Assessed on whether the classification is justified from flows rather than from what the system *felt* like, and on whether the second half names rows of the obligations table specifically rather than gesturing at "more testing".

Exercise 11 (hint). The easiest construction uses the consolidated academic definition [92], which requires only that information be exchanged and does not name a direction – so a rich monitoring platform satisfies it and fails the data-flow test. A good answer then picks the data-

flow test for the contract and says why: it is the one whose answer can be checked from a deployment diagram, and a contractual term that cannot be checked is not a term. A very good answer concedes the cost: the test says nothing about whether the twin is any good, which is why the contract also needs the credibility clauses a later chapter supplies.

Exercise 12 (hint). No solution, but a test. Read your three deliverables and ask, for each: could two engineers disagree about whether it has been met? If yes, it is still a sentence, not a deliverable. And check that one of the three says, in so many words, whether anything is written back to the line. That clause is the whole chapter.

3.11 Further Reading

- *The taxonomy itself, first-hand:* [51] is the source of the model/shadow/twin split and the paper to read if you read only one. For independent restatements that each add a nuance, [88], [60], [56] and [31]; for how often a paper's "twin" turns out to be a shadow, [101].
- *On there being no settled definition:* [1] for the twin/shadow boundary in manufacturing and [38] for the practitioner survey and the argument that a standard should fix it.
- *Model, simulation and simulator:* [48] traces the concept from classical modelling and simulation to twins and carries the workflow (conceptual model, executable model, verification, calibration) and NASA's definition; [45] and [47] open the book-length treatment this chapter's framing is closest to; [87] is the standards analysts' answer to "how is a twin different from simulation".
- *Definitions from outside the taxonomy:* [91] for ISO 23247's wording, [92] for the consolidation exercise and its lexicon, and [90] for the US National Academies' definition, which widens the referent and insists on prediction.
- *Closing the loop:* [89] for why the return arrow, and the autonomy it grants, is what separates a twin from a shadow, and [54] for the same loop read as a self-adaptive system.
- *Non-physical referents:* [34] for the "actual twin" phrase and [52] for conceptual referents; [93] for a twin of a software engineering process; [94] and [95] for the twin as software and TwinOps; [96] for DevOps on a building twin and [97] for a campus twin; [98] and [99] for hospital processes; [2] for supply chains; [100] for urban mobility with the human kept in the loop.
- *Types, by referent:* [21] and [101] for the published type schemes; [102], [81] and [103] on lifecycle twins; [104], [44], [105] and [14] on biological twins, read together so that the deployments and the aspirations stay distinct; [106], [107] and [108] on ecological and Earth-system twins; [109] for a twin of the human in a mobility system.
- *The hard cases:* [110] on what visualisation is for, and [111] for human-in-the-loop as a named twin style.
- *The rig's relatives:* [53] for the greenhouse exemplar whose twin manages its pumps through a feedback loop; it is the closest published relative of the four stages worked above.
- *In the rig itself:* open the controller's REST API documentation at `/docs` and find the four endpoints listed in the Classification section. Then find the one `PUT` among them. That single route is the boundary this chapter is about, and everything the rest of the book builds sits on one side of it or the other.

Chapter 4

Digital Twin Architecture

4.1 Before You Start

What you are assumed to know. You write software for a living, you are comfortable with HTTP APIs, JSON, a time-series or relational database, and reading Python you did not write, and you have read the previous three chapters. From Chapter 1 you have the six value patterns – *monitor, diagnose, predict, decide, control, certify* – and the lesson that a **digital twin (DT)** is bought to improve one decision about one specific real system, the **physical twin (PT)**. From Chapter 2 you have the plant rig: a potted plant, a pump and a Raspberry Pi controller whose REST API on port 8099 is the one boundary a twin is allowed to touch. From Chapter 3 you have three things this chapter leans on directly: the distinction between a model, a simulation and a simulator; the two-question data-flow test, which classifies a system by whether the PT's state reaches the **digital object (DO)** without a person carrying it and whether the DO's output reaches the PT the same way; and the reading of the return arrow as a control loop, with the authorisation, staleness, concurrency and fail-safe obligations that came with it. Each is recalled in a clause where it is used.

What you are not assumed to know. Anything about software architecture for digital twins in particular, or about the reference models a search for "digital twin architecture" turns up. Where this chapter needs a term from that literature it defines the term at first use and then keeps using it.

What the previous chapter left open, and what this one picks up. The previous chapter set aside three threads. It said that presentation – the dashboard, the chart – is *a service the twin offers*, and that a later chapter would cover it; this chapter does, in the Four Families of Services section. It said that a trained network is one candidate for *the model slot*, and that what changes when a learned model sits there was for later; this chapter treats the model slot as an architectural element, in the Seven Building Blocks section and again in the principles. And it said that a contract for a twin also needs *credibility clauses*; that thread stays open, because credibility is a question about how good a model is, and this chapter is about how the parts around the model are arranged. Data engineering – storage choice, retention, time-series design – also stays with the data chapters.

What this chapter covers. The previous chapter classified a system by its data flows and said nothing about what is inside the digital object. This chapter opens the box. It treats the digital object as what it is – a software system that runs for years, talks to hardware on one side and to people and other software on the other, and keeps a state that must stay close to the state of a real thing – and asks how a software engineer should arrange its parts. It names seven kinds of thing that live inside a twin; it sorts the twin's services into four families; it states eight principles and the patterns an architect will recognise; it gives one layered model to remember; it reads five published reference models for the one lesson each teaches; it shows how a twin is *described* so

that a platform can create and run it; and it ends by drawing the case study of Chapter 2 as an architecture.

What this chapter deliberately does not cover. Platforms and frameworks as products – what Azure Digital Twins, Eclipse Ditto or a Digital-Twin-as-a-Service platform actually look like to operate – which a later chapter takes up; here they appear only as the source of a configuration idea. Standards beyond the architectural content of one of them. How a model is built, calibrated or validated. Security, beyond the single principle of keeping the physical side and the digital side apart. Where the pieces run – edge, fog, cloud – beyond one paragraph.

Learning objectives. By the end of this chapter you will be able to:

1. **Place** the parts of a described or proposed twin onto the seven building blocks, and **explain** why state, and not data, sits at the centre.
2. **Classify** a twin's services into the four families, and **identify** which family delivers each of the six value patterns.
3. **Redraw** a twin as the three layers – physical interface, core, digital interface – **trace** one reading through the shadowing pipeline, and **state** in which lifecycle phase its state means anything.
4. **Read** a reference model you have not seen before, **map** its vocabulary onto the seven blocks, and **name** the one lesson it adds.
5. **Write** a twin's configuration at each of three rungs – a schema, a schema with relationships, a full configuration – and **argue** why it belongs in version control.

Every section that follows serves at least one of those five, and the exercises at the end say which.

4.2 A Digital Twin Is a Software System

Before naming a single component, settle what kind of thing is being built. A great deal of architectural confusion about twins comes from treating one as a dashboard with a database behind it, or as a simulation with a data feed bolted on. It is neither. It is a software system, and a particular kind of one, and four properties set it apart from the business software most of us have written.

It is long-running and stateful. A twin starts when the physical twin is commissioned and stops when the physical twin is retired, and between those two events it is never finished. A lift's twin runs for the thirty years the lift runs. That rules out the shape most data processing takes – a batch job that starts, computes, writes and exits – and rules in the shape of a service: a process that is always up, that holds a representation of the lift's condition at its centre, and that every other part reads from or writes to. The lifecycle literature makes the point from the other side: a twin has an operational lifecycle of its own, with named phases for starting, binding to its counterpart, running in step with it and shutting down, and that lifecycle is distinct from the phases of the physical asset it mirrors [81].

It is event-driven. Its inputs are readings and events that arrive when the physical world produces them, not when a user clicks. A wind turbine's twin does not wait for a request to learn that a bearing temperature rose; the reading arrives, and the twin's picture of the turbine changes. Commands leave the same way, as events towards the hardware. The frameworks built for twins are built this way – White Label Digital Twins (WLDT), the library the previous chapters borrowed the *physical adapter* from, is an event-driven engine in which physical and digital events ignite the processing that updates the state [82, 112] – and an engineer who arrives from request-response web services should expect the change of shape.

It is a control system whose controller lives in software. The previous chapter drew the return arrow as a control loop and attached obligations to it: authorisation, a staleness bound,

concurrency, a fail-safe behaviour. Once the controller runs *in the digital object*, those obligations stop being a checklist for a later increment and become architectural requirements. A pump's twin that is allowed to write a speed setpoint has to know how old its picture of the pump is before it writes, and has to have somewhere to refuse. Where that knowledge and that refusal live is an architectural decision, made in this chapter.

Its models are first-class artefacts with a lifecycle of their own. Ordinary information systems have code and data. A twin also has a model – a fitted curve, an exported simulation unit, a trained network – that describes how the asset behaves, that was produced by a different process from the code, that goes stale on a different schedule, and that may need to be swapped without rebuilding anything around it. The previous chapter's forward reference – "what changes when a learned model sits in the model slot" – has an architectural answer: nothing, if the slot was designed as a slot. The engineering literature treats the continuous, mutual evolution of the twin and its referent as the core problem of building twins at all [34], and the model is where most of that evolution lands.

4.2.1 What its architecture is judged on

A software architecture is judged against quality attributes, and a twin's is judged against two kinds. The first kind it shares with every information system. A systematic study of published digital-twin architectures started from the standard software-quality taxonomy and found performance efficiency, maintainability and compatibility – the ability to work alongside other systems – to be the three attributes architects mention most, ahead of eleven others that included scalability [59]. None of those is specific to twins, and a reader with your background already knows how to design for them.

The second kind is the twin's own: how faithfully, and how quickly, its state tracks the physical twin. The state is normally obtained by synchronising with the PT at discrete instants, and the processing that turns a reading into a state adds its own delay, so the twin's picture is always some distance behind the real thing; the middleware literature has proposed measuring that distance as an *entanglement* metric that combines how timely the state is with how complete it is [112, 113]. Whatever the metric, the attribute is the one that decides whether a twin is worth having, and it does not appear in the standard taxonomy at all. Every architectural decision below is, in the end, a decision about how to keep that distance small while the ordinary attributes stay acceptable.

The same study found that most published architectures are, at bottom, a division of functions with data flowing between them, and that the layered and service-oriented patterns are the two most used [59]. Hold both facts loosely for now. The next section proposes a division of functions; the Architectural and Design Principles section returns to the patterns.

4.3 Seven Building Blocks

Seven kinds of thing live inside a digital object, and it helps to name them separately even though real code mixes them. **Data** is what arrives from the physical twin and what the twin has recorded – time series, events, configuration. **State** is the twin's current answer to "what is the physical twin doing now": a small, structured, always-current representation that the rest of the twin reads. **Functions** are stateless computations – convert a unit, compute an efficiency, apply a rule. **Models** describe how the physical twin behaves. **Simulators** execute models. The **manager** creates, binds, starts, stops and restarts the twin and its adapters, and knows which lifecycle phase it is in. **Services** are what the twin offers outward: presentation, alerts, predictions, answers.

Two honest notes before the blocks are introduced one by one. First, the seven-block vocabulary is this book's own consolidation, not any one paper's. It joins two published views that describe the same thing from different ends. One is the *reusable-asset* view, in which a twin is composed from data, models, tools, functions and services that a platform hosts and reuses across twins rather than being built from scratch [114], and in which a twin is an assembly of services built upwards from models and data to analytics, visualisation and what-if analysis [79]. The other is the *state-and-shadowing* view, in which a twin is defined by a state that a shadowing process keeps aligned with the asset, behind a physical interface on one side and a digital interface on the other [82, 115]. The first view has no word for state; the second has no word for tools. The seven blocks name what the two views name between them. Second, the blocks are a decomposition of *responsibilities*, not of deployment units: a small twin runs all seven in one process, and a large one may run each in its own container, and the blocks are the same in both.

Figure 4.1 draws the seven around the one that the other six depend on. A single running example carries the whole section: a centrifugal pump in a water utility's pumping station, one of a few hundred the utility runs. Its data are flow, pressure, vibration and motor-power series; its state is the pump's current operating point, a health estimate and the last command sent to it; its functions are a unit conversion, an efficiency computation and a threshold rule; its model is the pump curve; its simulator is the code that steps the curve forward; its manager is what starts, binds, stops and restarts the twin and its adapters; and its services are everything that presents, alerts or answers a question on the pump's behalf.

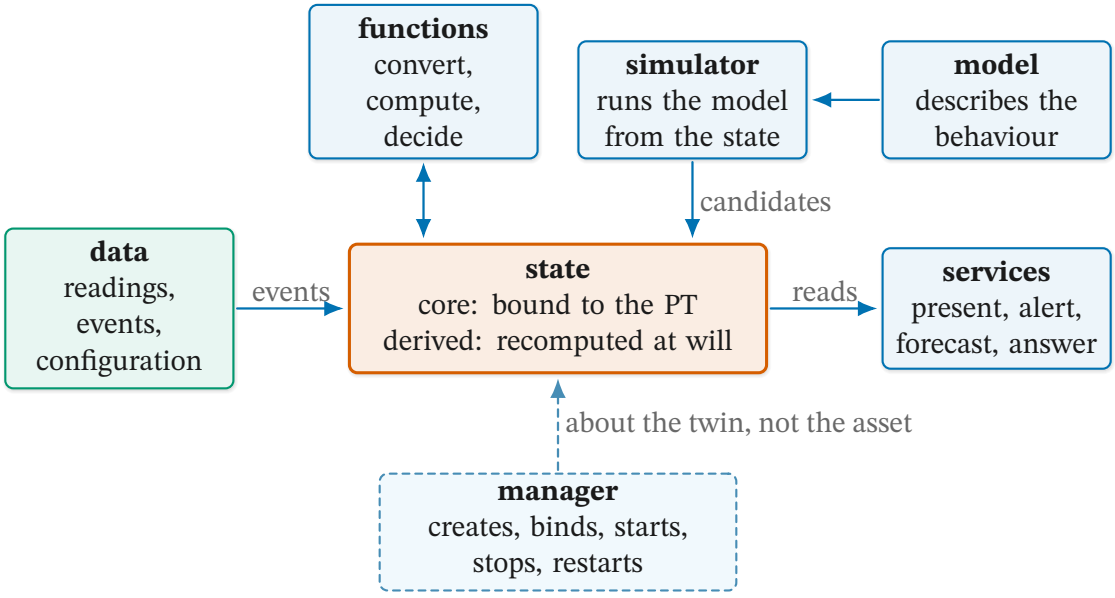


Figure 4.1: The seven building blocks, drawn around the one they all depend on. Data arrives and is recorded; functions and the model, run by a simulator, read and write the state; services read it; the manager stands outside the loop and is about the twin itself. Nothing reaches the state except through a block.

4.3.1 Data

Data is the raw material: readings and events from the physical twin, the history the twin has kept, and the configuration it runs under. Three kinds, then, and they are worth keeping apart. *Incoming* data is what the physical side delivers – the pump's flow reading every ten seconds, a "motor tripped" event when it happens. *Recorded* data is the history: those same readings kept

for weeks or years, in whatever store the utility has. *Configuration* is what the twin was told about itself and its pump – addresses, calibration constants, thresholds.

Data is not state. A five-year history of pressure readings tells you nothing about the pump *now* until something has computed a current estimate from it. That sentence is the reason the state block exists as a separate thing, and it is the sentence most often forgotten when a twin is designed database-first.

The twin reads and writes the data store, and it does not own it. Readings arriving from a physical twin are almost always kept in a time-series database, and several sources – the utility's maintenance system, an existing historian – may feed the same store [77]. Other software will read that store too, and the store will outlive any one version of the twin. Treat it as a neighbour the twin talks to, not as a component inside it. The standards side makes the same separation, if only by omission: the analysts of the manufacturing twin standard note that its reference architecture has no functional entity for data storage at all, and mentions a database only as one way for the twin and its users to exchange information [38]. Say "a time-series database" and stop there; which one, how it is laid out and how long it keeps things are the data chapters' business.

4.3.2 State

State is the twin's current belief about the physical twin. It is the one thing every other block depends on: functions read it and write it, models predict its next value, simulators produce candidate values for it, services present it and alert on it. Keeping state aligned with the physical twin is the shadowing process, and it is the twin's first job.

Three adjectives define it. It is *small*: a dozen fields for a pump, not five years of readings. It is *current*: it describes now, and it carries the time it describes. It is *structured*: every field has a name and a type, so that a service reading it does not have to parse anything. Here is the pump's:

```
{
  "twin_id": "pump-17",
  "lifecycle": "shadowed",
  "as_of": "2026-09-14T06:10:00Z",
  "core": {
    "flow_l_s": 41.2,
    "pressure_bar": 5.8,
    "vibration_mm_s": 2.1,
    "motor_power_kw": 37.5,
    "motor_status": "running",
    "last_command": {"speed_rpm": 1450, "at": "2026-09-14T05:00:00Z"}
  },
  "derived": {
    "efficiency": 0.71,
    "health": 0.93,
    "hours_to_next_service": 410,
    "operating_point": "on-curve"
  }
}
```

The two inner objects are not a formatting choice. Part of the state is bound directly to the physical twin – the last pressure reading, the motor's reported status, the last command that was actually sent – and no application may overwrite it. The rest is derived – the efficiency estimate, the health score, the predicted time to the next service – and may be recomputed at will. The Web

of Digital Twins model draws exactly this line: the properties and relationships of the *core* state are strictly bound to the physical asset and its evolution and cannot be changed directly by the application layer, while an *augmented* state, produced by additional functions, is kept separate from it precisely to mark the conceptual difference [115]. The distinction matters because it decides what a bug in a service can corrupt. A service with a wrong formula can produce a wrong health score; it cannot make the twin believe the motor is running when the motor said it stopped.

The `as_of` and `lifecycle` fields at the top are the state's own statement about itself. The first says how old the belief is, which the staleness obligation of the previous chapter needs. The second says whether the belief means anything at all, which the Layered Model section explains. An industrial ecosystem built on these ideas describes its twins as mirroring the state and capabilities of the real asset promptly, and calls that mirroring shadowing [84]; the two fields are how a consumer can tell whether the mirror is currently showing anything.

4.3.3 Functions

Functions are stateless computations that take data or state in and produce data or state out: convert a raw count to a pressure in bar, compute an efficiency from flow, head and motor power, decide whether a vibration threshold is crossed. They are the most reusable block – the same unit conversion serves every pump in the utility – and the easiest to test.

This is the software engineer's home turf, and it is the one block where ordinary unit testing applies without ceremony. A function has inputs and outputs and nothing else; a test gives it a reading and checks a number. The reusable-asset literature reaches the same conclusion from the platform side: functions are one of the two mandatory assets a twin is composed from, and the one that both services and tools reuse – the libraries a simulation tool depends on are function reuse by another name [114]. Three consequences for how you write them. Keep them pure: a function that reads a clock or a database is harder to test and impossible to share. Version them with the twin's configuration, because a threshold is a decision somebody made. And resist folding them into the services that call them, because the moment two services need the same efficiency computation, two copies will drift.

4.3.4 Models

The previous chapter pinned the word: a model is a description of how something behaves – equations, code, rules, a fitted set of parameters – and it is static; it does not run. Here the point is architectural. Architecturally a model is a *slot*, and the twin should not care what fills it: a fitted pump curve, a Functional Mock-up Unit (FMU) exported from a modelling tool, or a trained network. What the slot demands is an interface – give me a state and a horizon, return a predicted state – and the twin is designed against that interface, not against any particular model.

That is the answer to the previous chapter's open question. When a learned model replaces the pump curve, the adapters do not change, the state does not change, the services do not change; what changes is the thing behind one interface, and the evidence needed to trust it, which is the credibility chapter's problem rather than this one's. Two lines of published work show the slot in use. A manufacturing-cell twin integrates its behavioural models as FMUs through the Functional Mock-up Interface (FMI), administered by a *DT Manager* that stands between the twins, the physical system and the simulations [116]; and an open-source twin framework was re-defined so that FMI simulation services and machine-learning models could coexist behind the same twin, which is the slot filled two ways at once [55]. Neither system rebuilt its twin to change the model, and that is the property the slot exists to give you.

4.3.5 Simulators

A simulator executes a model from a given state over a given horizon. Separating the simulator from the model lets the twin swap either: the same solver runs the pump model and the pipe-network model, and the same pump model runs in a ten-line stepper and in a co-simulation engine.

In the pump's twin the simulator is a few lines of Python that step the curve forward. In an industrial twin it is a co-simulation engine that runs several models together, and the boundary between them is standardised: a co-simulation standard prescribes the communication interface and protocol between the simulation units and the orchestrator that steps them, so that units from different tools can be composed [117]. A shipboard crane's twin, for instance, is built as a co-simulation through the FMI standard [118]. What the twin needs from a simulator is the same in both cases: start from this state, run to this time, report.

Whatever the model, something has to *run* it, and that something is a piece of software the twin has to be given rather than a property of the model. An exported FMU is inert until an execution environment loads and steps it, which is why the surveyed twin frameworks ship one: a co-simulation toolchain separates the application that sets up a co-simulation scenario from the engine that runs it, with a third tool to export the FMUs and a fourth to stream data in and out [119], and a lighter alternative is a Python environment whose whole job is FMU execution [120]. Two later sections call such an execution environment a **tool** – the principles section, where it is one of the assets a twin is composed from, and the configuration section, where the twin's configuration has a place to name it. The word is worth carrying forward from here.

4.3.6 Manager

The manager is the block the literature names least consistently: "engine", "manager", "orchestrator", "lifecycle manager". Its jobs are consistent whatever it is called. It instantiates the twin, loads its configuration, starts and stops the adapters, binds the twin to its physical counterpart, tracks the lifecycle phase, restarts what fails, and exposes its own health.

The manager is the part of the twin that is about the twin rather than about the asset: it creates the twin, binds it to its physical counterpart, starts and stops the adapters, knows which phase of its lifecycle the twin is in, restarts what fails, and reports its own health. A twin that is running but not yet bound to its physical twin has state that means nothing; the manager is what knows the difference.

Three published managers show the range. WLDT's *engine* is a multi-threaded core that runs the adapters as independent workers and shapes the twin's behaviour around them [82]. The Web of Digital Twins gives the lifecycle it tracks: after start-up the twin is *operating but not bound*; a binding procedure attaches it to the physical asset; once bound it starts shadowing and reaches the *shadowed* state; an error during synchronisation drops it to *out of sync*; and at the end it is stopped [115]. And at platform scale, a Digital-Twin-as-a-Service reference architecture has a *DT Lifecycle Manager* component that manages each twin through creation, deployment, maintenance and termination, and notes that lifecycle support must either be built into a twin or offered to it by the platform through a standard interface [114]. Read the three together and "fleet" thinking enters: a manager for one pump and a manager for a utility's two hundred are the same block at two scales, and the configuration section says how one description serves both.

4.3.7 Services

A service is anything the twin offers to someone other than the physical twin: a chart, an alert, a forecast, an answer to a query, a recommended action. Every one of the six value patterns from the first chapter reaches its user as a service. The engineering literature groups a twin's services

into visualisation, monitoring and decision support [79]; the next section sorts them differently, by what each is centred on, because that is the sorting an architect needs. One paragraph here is enough, because the next section is about nothing else.

4.4 Four Families of Services

Services sort into four families by what they are centred on. A **data-centric** service answers questions about what was recorded: give me last week's pressures, export the maintenance log, replay Tuesday. A **function-centric** service runs a computation on request: forecast the next twelve hours, tell me what running at a higher speed would do, find the schedule that keeps the network in band. A **state-centric** service exposes the current state: the dashboard, a stream a client subscribes to, the endpoint another twin reads. An **alert-and-monitoring-centric** service watches state or data against a rule and tells someone – and, turned inward, watches the twin itself.

The classification is the author's, but each family has a home in the published treatments. The manufacturing twin standard's *application and service* sub-entity is described as providing simulation, analysis of data captured from the asset, and reporting of production status [87] – a function-centric, a data-centric and a state-centric service, in that order, under one heading. The engineering-of-digital-twins book devotes a part to services and groups them as visualisation, monitoring and decision support [79], with a chapter on the advanced ones – what-if simulation, fault diagnosis, predictive maintenance, reconfiguration – as the means by which a twin supports decisions and instigates change in its physical twin [30]. Read that list with the four families in hand: what-if and predictive maintenance are function-centric; fault diagnosis is alert-and-monitoring-centric with a function behind it; visualisation is state-centric; and reconfiguration is function-centric with a command path attached. The families are not a rival scheme. They are a sorting by *what the service reads*, and that is the question an architect asks first, because it decides which block the service is attached to.

Table 4.1 gives each family a generic instance and the value patterns it delivers.

Table 4.1: The four families of services, with a generic instance and the value patterns each delivers. *Monitor* is state-centric; *diagnose* and *predict* are function-centric with an alert attached; *decide* and *control* are function-centric, the second with a command path; *certify* is data-centric, because certification is an argument from records.

Family	Centred on	Pump or turbine instance	Value patterns delivered
Data-centric	recorded data	export last month's vibration; replay the day before the trip	certify; the evidence half of diagnose
Function-centric	a computation on request	forecast the next twelve hours; what happens at a higher speed; the schedule that keeps the network in band	diagnose, predict, decide; control, once a command path is attached
State-centric	the current state	the dashboard; a stream a client subscribes to; the endpoint the station's twin reads	monitor

Family	Centred on	Pump or turbine instance	Value patterns delivered
Alert-and-monitoring-centric	a rule over state or data, or the twin itself	vibration above threshold; a residual that stays positive; "no reading for ten minutes"	diagnose (the alert half); the twin's own health

Three remarks on the table.

The dashboard has a name now. The previous chapter said a dashboard is a shadow, not a twin. In this chapter's terms it is a state-centric service, and the sentence is a compliment: it names precisely what the dashboard is inside the architecture. A good state-centric service is the most-used service most twins have, and the previous chapter's observation that visualisation shows a person a shape in the data that a table hides [110] is a statement about how well a state-centric service does its job, not about whether the thing behind it is a twin.

A function-centric service is where the model earns its keep. Every forecast, what-if and optimisation runs the model through the simulator from the current state; the service is the request-response wrapper around that run. This is also why the model slot of the previous section has to be a slot: the function-centric services are written against "predict from this state over this horizon", and would all break if they were written against one model's own interface.

The fourth family has two halves, and the second is the one most designs leave out. Watching the *asset* through the twin – vibration over a threshold, a residual that stays positive – is what most people mean by alerting. Watching the *twin* is different, and deserves its own paragraph.

4.4.1 Monitoring the twin itself

A twin needs to be monitored as a piece of software, separately from monitoring the asset through it. A twin whose adapter has lost its connection shows a steady reading, and a steady reading is what a healthy asset on a quiet day also shows. Metrics on message arrival, shadowing latency and lifecycle phase are how the difference is seen.

This is ordinary observability, applied to an unusual program. The WLDT library ships a metrics manager for exactly this: a developer can define, measure and send to a local or remote collector all of a twin's metrics and logs, and the authors note the information is useful both for balancing load across twins and for detecting unexpected behaviour or performance degradation [82]. A later study from the same line of work goes further and treats metric-based observability as a lightweight way of detecting that something is wrong with a twin, showing that attacks on the twin's inputs leave measurable fingerprints across its metrics [113]. And a survey of twin-as-a-service platforms describes the platform side: collecting processor load, memory use and network flow from each twin instance, profiling them over time to establish a baseline, and reading a sudden shift – a twin instance suddenly consuming far more than usual – as an early sign that something needs attention [121]. For the pump: an alert on "no physical event received for ten minutes" catches the lost connection that the steady reading hid, and an alert on "lifecycle is not `shadowed`" catches the twin that restarted and never re-bound. Neither says anything about the pump. Both say something about whether anything the twin says about the pump can be believed.

4.5 Architectural and Design Principles

Eight principles, each stated as a rule with a consequence, and each grounded where the literature grounds it. Then the pattern vocabulary an architect will recognise, and what all of it costs.

1. Separate the physical side from the digital side: two interfaces, never one. A twin has two faces, and they are different. Towards the physical twin it speaks the protocols the hardware speaks and it must never lie about what it observed. Towards its consumers it speaks HTTP, Message Queuing Telemetry Transport (MQTT) or a message bus and offers a model of the asset, not a stream of register values. Collapsing the two into one interface is the most common early mistake, and the hardest to undo. The frameworks that get this right build it in: the twin's interaction with its two worlds is carried by a physical interface and a digital interface, each composed of independent adapters that can implement different protocols, and a physical event and a digital event start two different processes in the core – one deciding which changes of the asset's state to map into the twin's, the other propagating a request from an application down to the asset [112].

2. State is the single source of truth inside the twin. A service that wants the current pressure reads the state; it does not go to the database and re-derive it, because two services deriving the same quantity two ways will eventually disagree. This is the property a platform sells when it sells state management: one survey summarises what a twin platform such as Eclipse Ditto offers as state management with event-based notification of changes, "a single source of truth" for a physical asset [49]. The rule has a corollary from the State subsection: the core part of that truth is written only by shadowing, never by an application [115].

3. The model is a slot behind an interface. The twin is written against "predict the state at time t from the state now"; the fitted curve, the exported FMU and the trained network are three implementations of that interface, and swapping one for another must not touch the adapters, the state or the services. The two systems cited in the Models subsection are the existence proof: a manager that administers FMU models behind one interface [116], and a framework in which FMI services and learned models sit behind the same twin [55].

4. Events in, events out. Prefer an event-driven core to request-driven polling of the physical twin. A reading arrives and the state changes; the state changes and a digital event is announced; a service reacts to the event. Polling has its place – the previous chapters' rig is polled over HTTP, because that is what its API offers – but the twin's *core* should be written as if events arrive, so that a push-capable physical adapter can replace a polling one without the core noticing. The blueprint architectures are explicit that the components are decoupled and interact by events [112].

5. Build from reusable assets, and compose twins rather than hand-build each. Twins are best built by composing reusable assets – functions, models, tools, data – rather than written from scratch for each asset, and a platform that hosts those assets is how a team goes from one twin to a fleet. The reusable-asset chapter states the principle in one line – it is easier to create twins from reusable assets than to build from scratch – and names the assets: data, models, tools and functions, with functions and tools mandatory [114]. The pump utility is the case for it: the unit conversions, the pump-curve model and the stepper are the same for two hundred pumps, and only the configuration differs.

6. Make the twin observable as software. The Monitoring the Twin Itself subsection made the case; the principle is that observability is designed in, not added. Every adapter counts the events it handled, the core measures the time from physical event to consolidated state, and the manager exposes the lifecycle phase. The metrics facility WLDT builds into its library [82] is what this looks like in a framework; in a hand-built twin it is three counters and a gauge.

7. Make the lifecycle explicit. A twin knows whether it is bound and shadowing, and

says so. The lifecycle the Web of Digital Twins describes – operating but not bound, bound, shadowed, out of sync, stopped [115] – is refined by the middleware work into a state machine in which a twin that has lost the quality of its synchronisation drops to a *disentangled* state where it is unable to provide its intended function, and a finished twin remains accessible, detached from its asset, exposing its history and last state until it is dismissed [122]. The principle for your own twin is smaller than either: have a phase field, set it honestly, and let every service read it before it trusts the state.

8. Segregate and secure the physical-digital boundary. A compromised digital side can drive an actuator. The two-interface split of principle 1 is also the place to put the security boundary: the physical interface is reachable only from the twin, the digital interface is the only thing consumers reach, and nothing on the digital side can reach the hardware except through the core's command path. The security study of these architectures puts the reason plainly: the twin's dual cyber-physical nature exposes it to attacks at several levels – at the network, in the software, in the data – and an attack that reaches the core reaches the asset [113]. A later chapter has more to say about security; this principle is the one architectural decision it needs made now.

4.5.1 The pattern vocabulary

An architect arriving from enterprise software will want to know which named patterns twins use, and the answer is reassuring. The systematic study found that architectural solutions for twins draw on a catalogue of ten patterns, alone or in combination, and that the layered and the service-oriented patterns are the two most used; it also found that more than half of the published solutions are simple divisions of functions with data flowing between them, which is what a *reference model* is, as opposed to a reference architecture that maps those functions onto software elements [59]. There is a dedicated catalogue too – a set of digital twin architecture design patterns proposed to guide architecting twin-based systems, on the observation that designers were describing their twin configurations informally [56] – and the Industrial Internet Reference Architecture lists, among its architectural patterns, a "digital twin core as a middleware" pattern in which the twin core sits between the physical equipment and the industrial applications in a three-tier arrangement [19]. None of the three sources' diagrams is reproduced here; the point is that the words are the ones you know. The layered model of the next section is a layered pattern. The four service families are service orientation. The middleware pattern is principle 1 drawn as tiers.

4.5.2 What the principles cost

Every principle above adds a moving part that a script polling a sensor and writing a database does not have. Two interfaces instead of one. A state document instead of a query. An interface in front of the model. An event bus, or at least an event loop. A platform to host assets. Counters and gauges. A phase field and the code that keeps it honest. A network boundary. For a first increment on one pot – the previous chapters' Stage 2 shadow – most of this is disproportionate, and the honest architectural advice is to build the polling script and know which of the eight principles it violates. The principles pay when the second asset arrives, or the first command leaves. The demonstrator section shows both moments on the rig.

4.6 The Layered Model: Physical Interface, Core, Digital Interface

If you remember one picture from this chapter, remember this one. The digital object has three layers. The **physical interface** faces the physical twin: its adapters speak I2C, MODBUS,

MQTT, Open Platform Communications Unified Architecture (OPC UA) or HTTP to whatever the hardware exposes, and they carry physical events inward and physical actions outward. The **core** holds the model, the state and the shadowing function that turns each physical event into a new state. The **digital interface** faces everyone else: its adapters expose the state and the twin's digital events over HTTP, MQTT or a message bus and accept the invocations that services and other twins make.

The model comes from WLDT and the Web of Digital Twins work that grew from it, and it is running code rather than a diagram. WLDT is a Java library designed, in its authors' words, to maximise modularity, reusability and flexibility, in which a twin is an independent software agent built on a multi-threaded engine that runs communication adapters as workers and can be extended with new modules for new protocols [82]. The edge-oriented successor states the three parts as the physical interface, the model and the digital interface, with the physical interface responsible for capturing data from the asset and controlling physical actions, the model computing the twin's state, and the digital interface exposing events and accepting invocations [83]; a recent security study restates the same three from a software viewpoint as the physical interface for sensing and synchronisation, the core model managing shadowing and augmentation, and the digital interface ensuring observability and interoperability [113]. Three independent statements of one structure, from one line of work, all of them compiled and measured. Figure 4.2 draws it.

4.6.1 The shadowing pipeline

Shadowing is a pipeline. A raw physical event arrives and is normalised. The shadowing function combines it with the model and the current state to compute a candidate new state. Further functions may enrich that candidate with derived properties and relationships. The result is consolidated as the new state and announced as a digital event, which is what the digital adapters and the services react to. Figure 4.3 walks one reading through the five steps; the blueprint architecture they come from states them as five handler functions in exactly this order – a physical event handler that normalises the raw message, a shadowing handler that computes a candidate state from the event and the model, augmentation functions that enrich the candidate with more properties and relationships, a twin handler that consolidates the new state and computes the digital event, and a digital event handler that sends it to listeners through the digital interface [112]. The return direction – an action requested on the twin propagated down to the asset – is a shorter pipeline of three steps that normally involves no augmentation [112, 115].

Take the pump's reading through it. A message arrives on the MODBUS adapter: register 4021 holds the value 5812. *Normalise*: the adapter turns it into a physical event, "pressure sensor 2 reports 5.812 bar at 06:10:00", applying the unit-conversion function. *Shadow*: the shadowing function takes the event, the current state and the model and produces a candidate state whose core pressure is the new reading, and whose `as_of` is the event's time. *Augment*: the efficiency function recomputes the efficiency from the new pressure and the last flow and power; the health function updates the health score. *Consolidate*: the candidate becomes the state, and a digital event – "pump-17 state changed; pressure, efficiency, health" – is raised. *Deliver*: the HTTP digital adapter serves the new state to the next request, and the MQTT digital adapter publishes the event to the dashboard and the alerting service that subscribed to it. Every step is a function you can test, and the only step that knows what a MODBUS register is happens inside the physical adapter.

4.6.2 The lifecycle

A twin has a lifecycle, and its state is only meaningful in one phase of it. After start-up the twin is operating but not yet bound to its physical counterpart; binding attaches it; once bound and

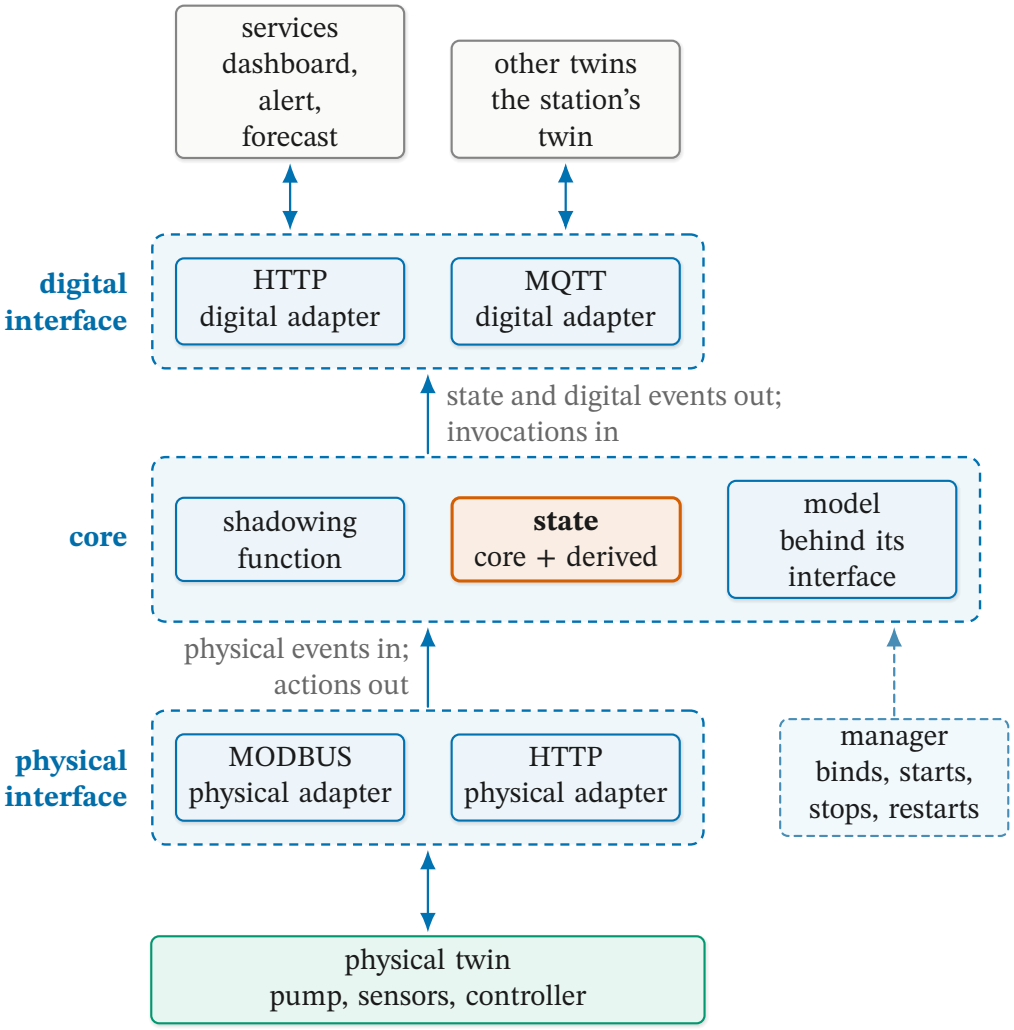


Figure 4.2: The three layers of a digital object. Physical events climb through the physical adapters into the core, where the shadowing function, the model and the state live; digital events and state leave through the digital adapters to services and other twins. Nothing crosses from the digital side to the hardware except through the core.

shadowing it is synchronised. State read from a twin in any other phase describes nothing real, and the digital interface should say so rather than serve a stale number.

The Web of Digital Twins draws the phases: after start-up the twin is *operating and not bound*, with all its modules active but no asset attached; the binding procedure connects the two; in the *bound* state the twin can handle events in both directions and starts shadowing; when synchronised in events and state it is *shadowed*; an error in synchronisation drops it to *out of sync*, where it can neither handle events nor align its state, until synchronisation is recovered; and it can be stopped, and finally dismissed [115]. The lifecycle work that followed is explicit about what the phases mean for the state: the computation of the twin's state should occur exclusively during the synchronised phase, the only one in which the twin actively mirrors its asset, while the other phases – start, unbound, bound, done, stopped – track the twin's own progress as software and do not involve synchronisation at all [81]. That is the justification for the `lifecycle` field in the pump's state, and for the rule that a state-centric service reads that field before it shows a number. A dashboard that shows a pressure of 5.8 bar over the word "unbound" is telling the truth. One that shows only the number is not.

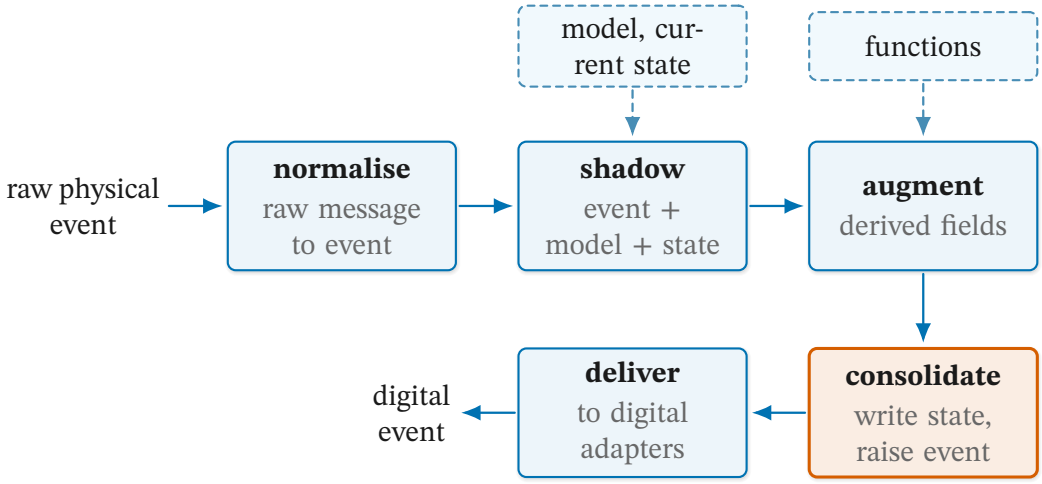


Figure 4.3: One reading through the shadowing pipeline. The raw message becomes a normalised event; the shadowing function turns the event, the model and the current state into a candidate; augmentation adds derived fields; consolidation writes the new state and raises a digital event; the digital adapters deliver it.

4.6.3 Every block on a layer

Table 4.2 places the seven building blocks onto the three layers. The mapping is the reason the two sections were written in this order: the blocks say *what* is inside a twin, the layers say *where*, and an architecture is answerable for both.

Table 4.2: Where each of the seven blocks lives in the three-layer model. Two blocks straddle a boundary on purpose: functions run in both the adapters and the core, and recorded data sits beside the core rather than inside it.

Block	Layer	Why there
Data (incoming)	physical interface	it arrives as physical events, in the hardware's own units
Data (recorded, configuration)	beside the core	the twin reads and writes the store; it does not own it
State	core	the thing shadowing keeps aligned, and the only thing services read
Functions	core, and inside adapters	normalisation lives in the physical adapter; augmentation lives in the core
Model	core	behind its interface, read by the shadowing function and the simulator
Simulator	core	runs the model from the state; a function-centric service calls it
Manager	outside all three	it starts, binds and stops the layers; it is about the twin, not the asset

Block	Layer	Why there
Services	digital interface, and beyond it	the digital adapters are the twin's side; a service is the consumer's

One more word on where the model came from, so you know what kind of claim it is. WLDT's design goals were a modular engine, adapters as pluggable modules and a small footprint at the edge, and its authors measured the library rather than describing it – the overhead it adds over a direct connection to the device for two protocols, its computational and memory cost on an edge node, and the lines of code a new adapter costs, all against the leading open-source alternative [82]. The Lessons from Five Reference Models section returns to why that matters: of the five models you are about to meet, this is the one that can be run.

4.7 Lessons from Five Reference Models

No single reference model has won, and a software engineer will meet at least five. They disagree in vocabulary more than in substance, and reading them side by side is how you learn which parts of a twin every author agrees on. This section gives each model the same treatment – what it says, in two or three sentences and without reproducing its figure; where it came from and for whom; and the one lesson a software engineer takes from it – and closes with a table that lines their vocabularies up against each other and against the seven blocks. A published unifying exercise has already mapped the data-flow taxonomy of the previous chapter, the five-dimension model, the manufacturing standard and the Asset Administration Shell onto a common set of concepts [60], and the table leans on it for the columns it covers.

One caution runs through all five: be clear which of them is a diagram and which is a system. Four are conceptual models – divisions of functionality with data flowing between them, which is what the architecture study found most published solutions to be [59]. One ships as a library. The lessons differ accordingly.

4.7.1 The five-dimension model

Grieves' original picture of a twin had three parts: a physical entity, a virtual entity and the connection between them. Tao and colleagues found that three could not accommodate the range of applications and technologies twins were being asked to serve, and extended it to five, adding *twin data* and *services*; in their formulation the twin is the tuple of physical entity, virtual entity, services, data and connection, where the virtual entity is a set of models and the connection ties all five parts together rather than joining only the first two [123]. It came from manufacturing research, it is a conceptual model, and it is the most widely repeated picture in the field; the same group later proposed makeTwin, a reference architecture for a twin *software platform* built on the five-dimension theory, with ten core functional modules for building models, processing data and the rest [124].

The five-dimension model adds twin data and services to Grieves' physical entity, virtual entity and connection, and makes the connections pairwise among all five. Its lesson for this chapter is that data and services are dimensions of a twin, not by-products of one. An architecture that treats the data as whatever happens to land in the database, and the services as whatever gets built on top, has left two of the five dimensions to chance.

4.7.2 ISO 23247

The manufacturing standard from the International Organization for Standardization (ISO) is the one reference model here that is also a standard, and this section keeps to its architectural content; a later chapter covers the rest. Its reference architecture arranges a twin as four domains – the observable manufacturing elements, device communication, the digital twin and the user – with entities in the top three. The *digital twin entity* is split into three sub-entities: an *operation and management* sub-entity that maintains information about the manufacturing elements, including their digital modelling, presentation and synchronisation; an *application and service* sub-entity that provides simulation, analysis and reporting; and a *resource access and interchange* sub-entity that exchanges information with the user entity, with interoperability support. The *device communication entity* has a data-collection sub-entity and a device-control sub-entity, and a *cross-system entity* sits across the domains to provide data translation, data assurance and security support [87]. A companion paper walks use-case scenarios through the same functional-entity view [125].

ISO 23247 arranges a manufacturing twin into four domains – the observable manufacturing elements, device communication, the digital twin itself and the user – and splits the digital twin entity into operation and management, application and service, and resource access and interchange. Its lesson is that a functional decomposition is a list of responsibilities, and an architecture is answerable for where each one lives. Take the standard's list and ask, of your own design, which component does the data collection, which does the device control, which does the synchronisation, which does the reporting, which does the data translation; a responsibility with no home is a bug that has not happened yet. The study of how published manufacturing twins align with the standard found one gap worth knowing: the standard has no functional entity for data storage, though most real architectures have one [38] – which is the seven blocks' Data subsection arriving from the standards side.

4.7.3 The Digital Twin Consortium platform stack

The Digital Twin Consortium (DTC) is an industry body, and its Platform Stack Architectural Framework is written for people deciding what to buy and build rather than for researchers. It names six elements: security, trust and governance as the foundation for everything else; the real world the twin is instantiated from and synchronised with; a platform of information technology (IT) and operational technology (OT) – the IT/OT platform – on which the twin system is implemented; the virtual representation; service interfaces wrapped around it for integration and interoperability; and applications, where the twin's value is realised [126]. The consortium's own summary of the framework lists the primary layers as the IT/OT platform, the virtual representation, the integration and synchronisation services, and an application layer for insights [127], and its glossary defines the IT/OT platform as the set of information- and operational-technology infrastructure and services on which the twin system's subsystems are implemented – a software platform and tooling stack, platform APIs, orchestration of the low-level infrastructure, compute and storage [128]. A separate consortium paper takes the systems-of-systems view, proposing an interoperability framework for plugging multi-vendor services into a system of twins [129].

The Digital Twin Consortium's platform stack puts security, trust and governance under everything, then the real world, an IT/OT platform, the virtual representation, service interfaces for integration and synchronisation, and applications on top. Its lesson is that the virtual representation is only one layer of a twin system, and that the service interfaces around it are where interoperability is won or lost. A team that has built a beautiful virtual representation and no service interfaces has built a model, not a twin system; a model-driven engineering group that adopted the framework describes the IT/OT platform as the virtualisation foundation the twin

system operates on [64], which is the same lesson in different words.

4.7.4 White Label Digital Twins

WLDT has been this chapter's spine since the Layered Model section, and here it gets one subsection on equal footing with the others. It is an open-source, event-driven Java library for building twins for the Internet of Things (IoT): a twin is a core with pluggable physical and digital adapters, run by a multi-threaded engine, and the library was evaluated at the edge for overhead, cost and development effort against the leading open-source alternative [82]. It is also the basis for the Web of Digital Twins ecosystem work, which uses it as one of the technologies its adapters wrap [130], and a survey of twin frameworks describes it as running at the edge or in the cloud and supporting a wide range of protocols [49]. The middleware work built on it observes what it lacks: a management layer for deploying and orchestrating many twins in a structured, observable way [122] – which is this chapter's manager block at fleet scale, and the next model's subject.

White Label Digital Twins is the one of the five that runs: an event-driven framework in which a twin is a core with pluggable physical and digital adapters, and its lesson is that an architecture expressed as a framework can be measured for overhead, latency and scale, where a conceptual model can only be argued about. When two reference models disagree, the one you can compile settles the argument for your case in an afternoon.

4.7.5 The six-dimension model and the interface controller

The newest of the five is a 2025 proposal for complex industrial systems. It starts from the five-dimension model and adds a sixth dimension, an *interface controller*, on the argument that interface management is the difficulty in twin platforms for systems with many interacting components, devices and processes; the accompanying platform design pairs the controller with a dual-core data architecture that combines a relational and a time-series database [85]. It is one recent paper, and its adoption is not something this chapter can report on; its idea is what matters here.

The six-dimension model adds an interface controller to the five dimensions, on the argument that in a complex industrial system the interfaces between physical entities, virtual entities, data and services are numerous enough to need managing in their own right. Its lesson is the one this chapter's manager block already draws: past a handful of adapters, interfaces are a first-class concern. WLDT's engine running its adapters as workers is the same lesson at the scale of one twin; the interface controller is it at the scale of a plant.

4.7.6 Reading the five together

Table 4.3 sets the five side by side. Its rows are this chapter's seven blocks plus the two things every model has that the blocks leave outside the digital object, the physical twin and the connection; its columns are the five models. The five-dimension and ISO columns follow the published cross-mapping, which lines up "physical entity" with "observable manufacturing element", "virtual entity" with "digital representation", "twin data" with the data-collection entity and "connection" with device communication [60]; the other three columns are filled from the models' own papers. A dash means the model has no named part for that row – not that a twin built to it would lack the thing, only that the model does not name it.

Table 4.3: Five reference models against the seven blocks. Every column names a physical thing, a digital representation, a connection and data; services are named by four, a lifecycle or manager by three, and only the two newest name the interfaces as a thing to be managed.

This chapter	Five-dimension	ISO 23247	DTC platform stack	WLDT	Six-dimension
Physical twin	physical entity	observable manufacturing element	the real world	physical asset	physical entity
Connection	connection (among all five)	device communication entity	synchronisation, in the service interfaces	physical adapters and shadowing	connection, via the interface controller
Data	twin data	data collection sub-entity; no storage entity	IT/OT platform (compute and storage)	physical events	data, in a dual relational and time-series store
State	–	digital representation, kept synchronised by operation and management	virtual representation	twin state: core and augmented	–
Functions	–	application and service (analysis)	–	shadowing and augmentation functions	–
Models	virtual entity (a set of models)	digital modelling, in operation and management	virtual representation	model, in the core	virtual entity
Simulators	–	application and service (simulation)	tooling stack, in the IT/OT platform	–	–
Manager	–	operation and management; cross-system entity	orchestration, in the IT/OT platform	engine and lifecycle	interface controller
Services	services	application and service; resource access and interchange	service interfaces; applications	digital adapters, to applications	services

Set side by side, the five models agree on four things – a physical thing, a digital representa-

tion of it, a connection between the two, and data – and diverge on whether services, lifecycle and the interfaces themselves are named parts of the twin. This chapter's seven blocks are a consolidation of what they name between them, not a sixth model. Two readings of the table are worth making. Read across the State row and notice that the two conceptual models from manufacturing have no word for it: they name the models and the data and leave the current belief implicit, which is exactly the omission the State subsection warned against. Read down the WLDT column and notice that it is the only one with entries for state, functions and manager and none for simulator – a framework names what it implements, and it implements the loop, not the solver. The Asset Administration Shell, which the cross-mapping also covers, is a different kind of thing again: a hierarchy of data models called submodels, intended as the single source of truth about an asset across its lifecycle [131], and so a candidate for the schema rung of the next section rather than a sixth column here.

4.8 Digital Twin Configuration

An architecture says what the parts are; a configuration says which parts this twin has, where they come from and how they connect. Once a team has more than one twin, the configuration is the artefact they actually edit, review and deploy, and it deserves the same care as code. This section is the bridge from the architecture above to something a team can check into a repository. It climbs three rungs, each a subsection, each with pointers rather than a tutorial – the platforms chapter does the hands-on – and ties each rung back to the seven blocks: a schema describes state; relationships describe how states refer to each other; a full configuration describes all seven and where they run. Figure 4.4 is the ladder.

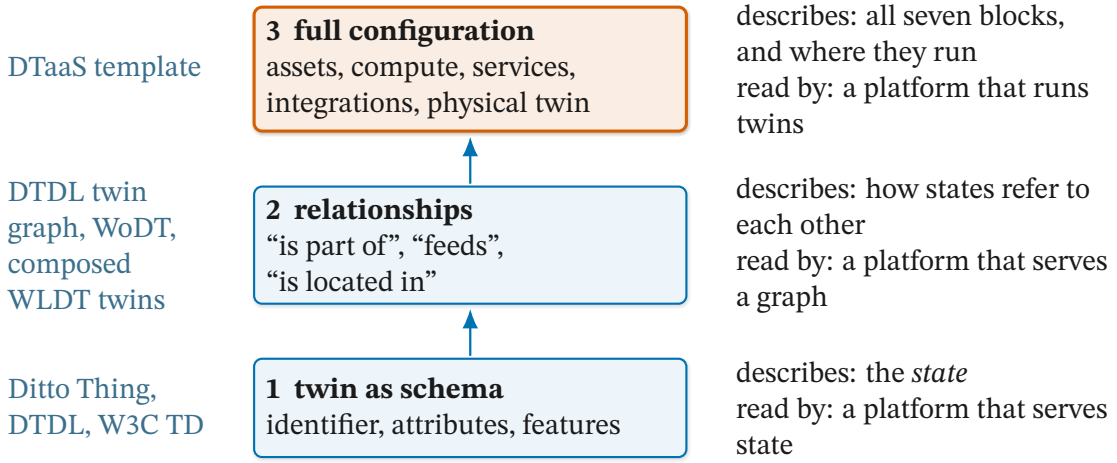


Figure 4.4: The three rungs of a twin's configuration. Each rung describes more of the twin than the one below it, and each is what a different kind of platform reads: a schema is enough for a platform that serves state, relationships are enough for one that serves a graph, and a full configuration is what a platform that creates and runs twins needs.

The simplest configuration describes the twin as a schema: a document that names the twin, its attributes and its features, from which a platform serves the state. Anything that can describe state this way can be served by anything that reads the schema, and that is the interoperability argument for having one. A twin's schema should be able to name its relationships to other twins. A pump belongs to a station, a station feeds a district, a district is part of a network, and a consumer who reaches one twin should be able to navigate to the others. Relationships are what turn a set of twins into a graph. A full configuration describes the whole twin, not only its state:

the assets it is composed from, the compute it needs, the services it exposes, the external systems it integrates with and the connection to its physical twin. A minimal configuration names the assets and the compute and is enough to run a twin without a physical counterpart; the physical-twin section is what closes the loop.

4.8.1 Twin as schema

In the Eclipse stack a twin is declared as a *Thing*: an identifier, an access policy, a set of static attributes and a set of features whose properties change as the physical twin does. The platform reads the declaration and serves the state; the developer never writes the endpoint. The survey of open-source twin frameworks describes the workflow – model the Thing, express it as a JSON document, store it in the platform, and then read and write its attributes and features through whichever of the platform's communication mechanisms suits [119] – and a framework built on the same platform describes the Thing as belonging to a namespace and consisting of an identifier, attributes for the static part of the entity and features for the dynamic part [132]. Here is the pump as a Thing, in ten lines of our own:

```
{
  "thingId": "utility.north:pump-17",
  "policyId": "utility.north:operators",
  "attributes": {"model": "NK 65-200", "station": "north-1"},
  "features": {
    "hydraulics": {"properties": {"flow_l_s": 41.2, "pressure_bar": 5.8}},
    "motor":      {"properties": {"power_kw": 37.5, "status": "running"}},
    "health":     {"properties": {"score": 0.93, "hours_to_service": 410}}
  }
}
```

Compare it with the state document of the Seven Building Blocks section and the correspondence is one-to-one: the features are the state, the attributes are the configuration the twin was told about itself, and the policy is who may read and write. That is why the schema rung is enough for a platform whose job is to serve state.

Ditto's Thing is one of three schema languages a reader will meet. Azure Digital Twins uses the Digital Twin Definition Language (DTDL), and the World Wide Web Consortium (W3C) has the Thing Description from its Web of Things work. A recent comparison of *digital twin descriptions* lists the three most prominent as the Asset Administration Shell, the W3C Thing Description and DTDL, and observes that they take different approaches to what a description contains – the first includes all the information in the document itself, the second says where the data can be accessed, and the third describes the model and leaves the values to the platform [133]. Which you meet depends on which platform your organisation bought.

A schema captures what the state looks like and nothing else. It says nothing about how the state is computed, in which lifecycle phase it is valid, or on which machine the twin runs. Look again at the Thing above: there is no pump curve in it, no shadowing function, no adapter, no process. A platform that reads only schemas is a platform for serving state, and it is a perfectly good thing to be; it is not a platform for running twins.

4.8.2 Relationships

A flat list of twins is not enough. A relationship is a typed, directed link from one twin to another – "is part of", "feeds", "is located in" – and a platform that stores relationships lets a consumer

navigate from any twin to the twins it depends on. Composed twins built from relationships are how a plant-level view is assembled from machine-level twins without copying their state.

You will meet relationships in three places. In DTDL, a model defines a twin's properties, its commands and its relationships with other twins, and the platform instantiates models and stores the last values of their properties in what it calls a *twin graph* [134]. In the Web of Digital Twins model, a twin is described by properties – observable attributes of the asset – relationships to other assets, expressed as links to their twins, and events, and the relationships can change dynamically as properties can [115, 135]. And in the hierarchical composition work built on WLDT, machine-level twins are composed into department- and plant-level views through relationships, so that resources group into departments, departments into business units and business units into plants, with a composed metric such as overall equipment effectiveness computed at each level, and the ecosystem can be navigated without prior knowledge of its structure [84]. A toolchain that wraps several platforms behind one interface records the practical consequence: of the platforms it compares, some store relationships between twins and some do not, and the ones that do are the ones whose twins can be reached from one another rather than only listed [130].

Composition and aggregation are the two shapes a relationship takes, and they are worth separating in one paragraph. *Composition* says one twin is made of others – the station twin is composed of its pumps – and the composed twin's state is derived from theirs: the station's total flow is the sum of the pumps'. *Aggregation* says one twin refers to another without owning it – the pump twin refers to the tariff twin that prices its electricity – and neither state is derived from the other. The first justifies recomputing the parent when a child changes; the second justifies only a link. A schema language that can say which is which saves the platform from guessing.

4.8.3 Full configuration

A full configuration has a section for each thing the twin is made of – its assets, its compute, its services, its external integrations and its physical twin – and a platform is responsible for generating, verifying, deploying and modifying such documents, including the credentials no human should type.

The template comes from the Digital Twin as a Service proposal, and it is stated as a document with a version, an `assets` section whose first entry is the twin's own configuration and whose others are its models, data, functions and tools, a `compute` section naming the host kind and the software image, a `dt_services` section with internal services and their auto-generated credentials and external services with their ports and hostnames, and a `physical_twin` section for the information that comes from the physical twin to the digital one. Its authors are explicit about the division of labour: the template is necessarily generic, so any platform must provide automation for generating, verifying, modifying and deploying configurations; the service credentials are auto-generated by the platform to the relevant security standards; a minimal twin is specified with only the assets' configuration and the compute section, and that minimal configuration is sufficient to run it; the physical-twin section is needed only for a real-time feedback loop with the PT; and the services section is filled only if internal or external services are required [114]. The case studies built on the same platform show the shape filled in: an incubator twin's configuration has sections for the message broker, the time-series database, the physical twin's initial controller parameters, and the digital twin's own service-specific settings and per-model parameters [46].

Map the sections onto the seven blocks and the configuration is the architecture written down. `assets` names the models, the functions, the tools that run the models – the simulators – and the data connections. `compute` is where the manager will put it. `dt_services` is the digital interface and what sits behind it. `physical_twin` is the physical interface. State is the one block

with no section, and rightly: state is what the running twin produces, not what it is told.

Two ideas make one configuration serve a fleet. The first is the distinction between a *twin class* and a *twin instance*: a physical product model is a class from which many physical instances are manufactured, each with its own identity; a twin class is developed in tandem with the product, and each manufactured instance gets its own twin instance, the two evolving together [114]. The utility's two hundred pumps are two hundred instances of a handful of classes, and the configuration of a class differs from the configuration of an instance only in the addresses, the calibration constants and the identity. The second idea is drift. Written declaratively and kept in version control, the configuration is also a drift detector: the platform can compare the twin that is running against the twin that was described and report the difference. The GitOps literature names the benefits of a declarative description of infrastructure with a process that makes the environment match it – faster deployment, declarative configuration, configuration drift detection, fast error recovery – and a recent proposal applies exactly that discipline to managing the lifecycle of twins described in a static definition language [136]. That is the reason the configuration belongs in a repository beside the code: not for tidiness, but because a running twin that no longer matches its description is a fault, and a versioned description is what makes the fault visible.

4.9 Architecture of the Plant Demonstrator

Everything above used a pump, a turbine or a ward. This section applies the chapter, in order, to the one system the previous chapters put on the desk, and it is the only place the plant demonstrator appears in this chapter. It is the chapter's fully worked example; the Faded Examples section then takes the scaffolding away.

Recall what Chapter 2 established. The physical twin is the pot, plant, soil, pump, tubing and the Raspberry Pi controller. The controller exposes a REST API on port 8099 with seven endpoints that read and one, `PUT /actuation/{unit}/update_schedule`, that writes; it stores the moisture and water-level series in InfluxDB; and its behaviour was written down as two equations, one for drying and one for dose response, from which the twin predicts the next reading $\hat{m}$, compares it with the observed m_{obs}, and calls the difference the residual $r = \hat{m} - m_{\text{obs}}$. The dose rule $V = \min\big(V_{\max},\ k\,(m_{\text{t}} - m)\big)$ says how much water to ask for. Recall also the previous chapter's two stages that matter here: Stage 2, a polling fault detector whose alert a technician acts on – a shadow – and Stage 3, the same service writing schedules through the one `PUT` – a twin.

The rig is a small member of a recognised family. The greenhouse exemplar the previous chapter cited has, as the core of its software architecture, an asset model of the physical system and simulation models used for prediction, arranged so that the software can adapt when the physical system's composition changes [53]. Nothing below is a toy argument about a toy.

4.9.1 The seven blocks on the rig

At Stage 2 the rig's digital object is one physical adapter, one state document, one model, one simulator, three functions, a manager and three services. Table 4.4 places them.

Table 4.4: The seven blocks placed on the rig at Stage 2. Four of the seven were already on the desk under other names; the state, the manager and the naming of the services are what this chapter adds.

Block	On the rig	Where it came from
Data	the moisture and water-level series in InfluxDB; the watering events; the controller's own configuration	Chapter 2
State	a JSON document holding the current estimate $\hat{m}$, the observed m_{obs}, the residual r, the schedule in force, the last dose and its time, the lifecycle phase	this chapter
Functions	raw count to moisture; the residual; the dose rule	Chapter 2's conversions and rules
Model	the two equations, drying and dose response, with d and g	Chapter 2
Simulator	the dozen-line Python stepper that advances the model T_c hours	Chapter 3
Manager	what binds one twin to one pot, starts its adapter, and knows whether it is shadowing	this chapter
Services	a state-centric dashboard; an alert on the residual; a function-centric forecast	Chapters 1 and 3's increments, named

Two rows deserve a sentence. The state row is new, and it is the chapter's centre applied: at Stage 2 the fault detector kept $\hat{m}$ in a variable between polls, and this chapter says that variable is the twin's state, deserves a document of its own with an `as_of` and a lifecycle field, and is what the dashboard reads. The functions row is old under new names: the conversion from the probe's raw count to a percentage, the residual and the dose rule were all written in Chapter 2 as equations; here they are three stateless, testable functions, and the dose rule is the one of the three that Stage 3 will let act.

4.9.2 The three layers on the rig

Figure 4.5 draws the rig's twin as the three layers. The physical interface of the previous chapters – the rig's REST API – is the boundary at which a physical adapter begins. Chapter 2 described that interface from the hardware's side; this section describes what sits on the other side of it. The physical adapter is a small piece of code that speaks HTTP to port 8099: at Stage 2 it knows the seven `GET`s, and it turns each response into a physical event – "plant_1 moisture 41.2 per cent at 10:31:40". The core holds the state document, the two equations behind the model interface, the stepper, and a shadowing function that, on each new reading, writes m_{obs}, computes r, and, when a dose was due, asks the stepper for the next $\hat{m}$. The digital interface is one HTTP adapter of the twin's own – not the controller's – that serves the state and raises an event when it changes. The three services sit beyond it.

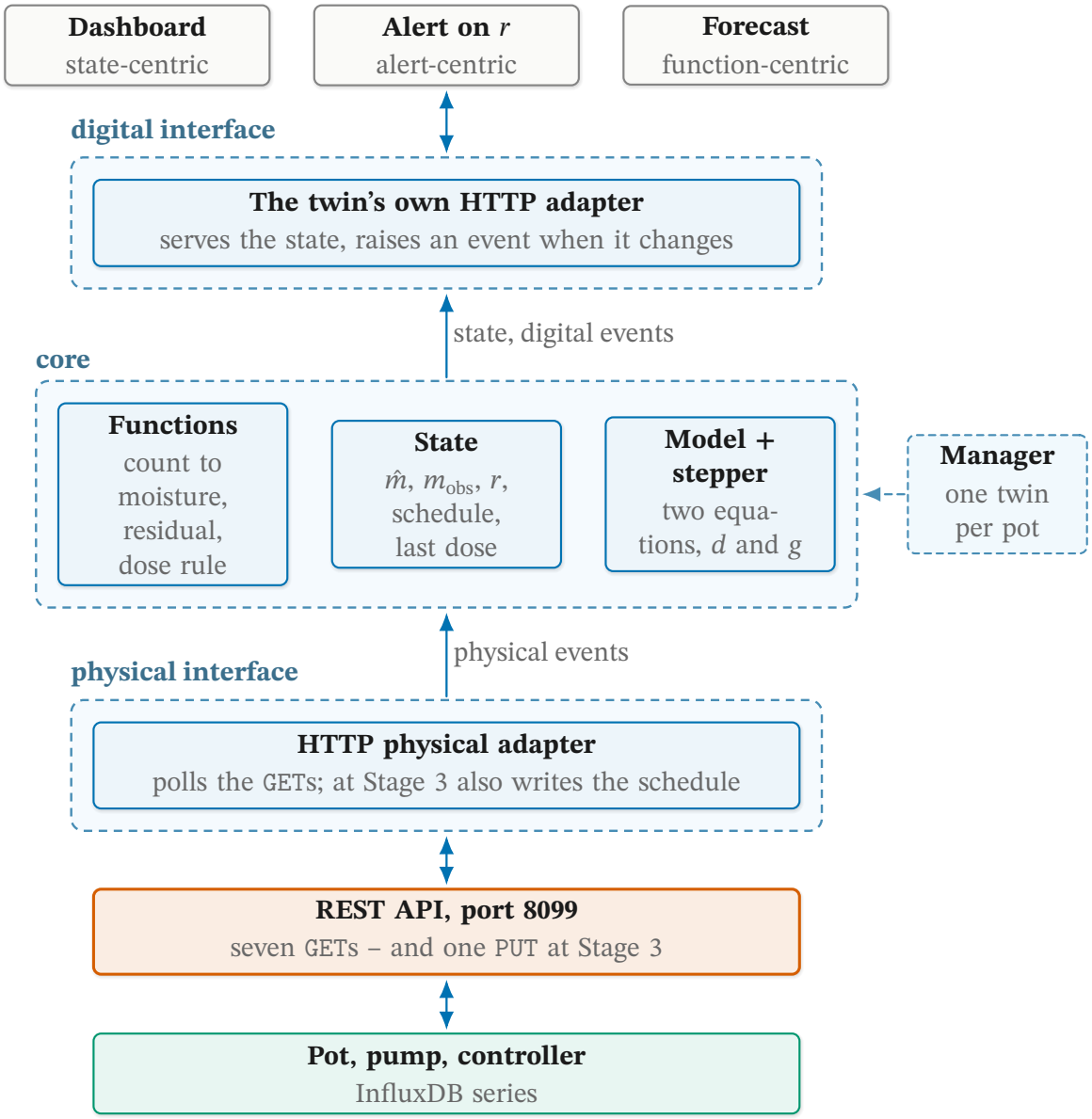


Figure 4.5: The rig's twin as three layers. The controller's REST API is where the physical adapter begins; the core holds the state, the two equations and the stepper; the twin's own HTTP adapter serves the state to a dashboard, an alert and a forecast. At Stage 3 nothing is added to the layers – the physical adapter gains the one `PUT`.

Trace one reading through the pipeline of the Layered Model section. The physical adapter polls `GET /sensing/plant_1/moisture?limit=1` and normalises the JSON object into a physical event. The shadowing function takes the event and the current state: it stores the reading as m_{obs}, computes r against the $\hat{m}$ it wrote down last time, and sets `as_of` to the reading's time. Augmentation recomputes the derived fields – time in the moisture band, water per day – from the recorded series. Consolidation writes the state and raises a digital event. The dashboard redraws; the alerting service checks whether $r > 0$ has held for the last three checks. Six steps, and the only one that knows the controller's JSON layout is the first.

4.9.3 The four families on the rig

The dashboard is state-centric: it reads the state document and shows m_{obs}, $\hat{m}$, the schedule in force and the lifecycle phase. The alert is alert-and-monitoring-centric, pointed at the asset: a residual that stays positive says less water is arriving than was asked for, which was Chapter 1's *diagnose* increment. The forecast is function-centric: given the state and a horizon, run the stepper and return the predicted moisture curve, which is Chapter 1's *predict*. The fourth family's inward half is the one the earlier chapters did not have: a "no reading for two check intervals" alert and a "lifecycle is not shadowed" alert, which say nothing about the plant and everything about whether the dashboard can be believed. A data-centric service – export last month's moisture and doses – is trivial to add, because the recorded data already sits in InfluxDB and the twin does not own it.

4.9.4 The principles applied, and two failures they predict

Walk the eight principles down the rig and two of them predict specific failures, which is what a principle is for.

State is the single source of truth. Suppose a second developer writes the dashboard against InfluxDB directly, re-deriving the current moisture from the last row of the series, while the alerting service reads the state document. On a quiet day both agree. On the day the physical adapter's connection drops, the dashboard keeps showing the last row – which is still the last row – while the state document's `as_of` stops advancing and its lifecycle field falls to out of sync. Two services, two truths, and the one a person looks at is the one that lies. The fix is the principle: the dashboard reads the state, and the state carries its own age.

Make the lifecycle explicit, and make the twin observable. Suppose the twin restarts overnight and the controller is unreachable for an hour. Without a lifecycle field the twin comes back with an empty state, polls, fails, and serves whatever it last had – a plausible number with nothing to say it is stale. With the field, the manager holds the twin at operating-but-not-bound until the first successful poll, the dashboard shows the word rather than a number, and the inward alert fires. The lifecycle work cited earlier is the general form of this story: the state is computed only while synchronised, and every other phase is the twin's own business [81].

The remaining principles either hold trivially on the rig – the two interfaces are two different HTTP conversations on two different ports; the model is two equations behind an interface the forecast service already assumes – or wait for the second pot: reusable assets and composition become worth their cost when a second twin runs the same functions against a second unit.

4.9.5 A configuration for the rig

The rig's configuration fits the full-configuration template in a page: its assets are two equations, one stepper, three functions and one time-series connection; its compute is one small process; its services are a dashboard, an alert and a forecast; and its physical-twin section is the address of the REST API and, at Stage 3, the one endpoint the twin is allowed to write. Here it is, in the template's shape and in our own words; the platforms chapter shows what a real platform does with a file like it.

```
version: 0.1
twin: plant_1

assets:
  config:    {location: file, path: plant_1.toml}
  models:    [drying.py, dose_response.py]      # d, g as parameters
```

```
  functions: [count_to_moisture.py, residual.py, dose_rule.py]
  tools:     [stepper.py]                          # runs the two models
  data:      {kind: influxdb, bucket: plant_controller}

compute:
  host: isolated host                              # one small process
  software-id: python-3.12

dt_services:
  internal:
    dashboard: {kind: state-centric,    port: 8100}
    alert:     {kind: alert-centric,    rule: "r > 0 for 3 checks"}
    forecast:  {kind: function-centric, horizon_hours: 24}
  external: []

physical_twin:
  api: http://plant-controller.local:8099
  reads:  [/sensing/plant_1/moisture, /actuation/plant_1/watering_events,
           /actuation/plant_1/show_schedule]
  writes: []                                       # stage 3: one PUT
```

Read it against the template's own rules [114]. The `assets` and `compute` sections alone would run the twin against recorded data with no pot attached – the minimal configuration, and a useful thing to have for testing. The `physical_twin` section is what closes the loop, and its `writes` list is empty at Stage 2 on purpose. The credentials for the time-series database are not in the file, because the platform generates them; a configuration in version control must never contain a secret.

4.9.6 Stage 3: one adapter gains a second direction

At Stage 3 nothing is added to the layers. One physical adapter gains a second direction, `PUT /actuation/{unit}/update_schedule`, and that is where every obligation of the previous chapter's command path is implemented. In the configuration above, `writes` acquires one entry. In the code, the physical adapter gains one method, and the shadowing function's return path – the three-step pipeline the Layered Model section mentioned – gains a command path box: is the `as_of` of the state younger than the staleness bound; is the dose under $V_{\max}$; did `GET /actuation/plant_1/show_schedule` return what the twin last wrote, or has a technician edited it; and if any check fails, keep the last safe schedule and raise the inward alert. The previous chapter listed those five obligations as distributed-systems problems wearing physical consequences. This chapter says which box they go in, and it is a box on the physical-interface side of the core, because the command path is the last thing that runs before a physical action leaves.

4.9.7 Deployment, in one paragraph

Where does it run? On the same Raspberry Pi as the controller, or on a laptop beside it, one process is enough for one pot – the twin is a few hundred lines and the stepper is a dozen. The question that matters arrives with the greenhouse: one process per pot, or one process for the greenhouse with an adapter per pot? The six-dimension model's lesson decides it. Past a handful of adapters the interfaces are a thing to manage in their own right [85], so a greenhouse of three pots is one process with three adapters and one manager, and a greenhouse of thirty is thirty twin instances of one twin class under one manager, each with its own state and its own lifecycle. The

configuration above is already written as an instance – `plant_1` – of a class; the thirty differ from it in one line.

4.10 Faded Examples

Three, decreasing in how much is worked, none of them the demonstrator. For each, place the blocks, draw the layers, and say what the architecture predicts.

4.10.1 Example 1, worked in full: the offshore turbine anomaly service

The previous chapter's second faded example: vibration and temperature stream from an offshore wind turbine to shore; a service compares them against a model of that turbine and raises a work order when the residual exceeds a threshold; a planner schedules a vessel. It was classified a shadow. Here it is redrawn as blocks and layers.

Blocks. Data: the vibration and temperature streams and their history. State: the turbine's current operating point and its residual. Functions: unit conversion and the threshold rule. Model: the turbine's expected-vibration model. Simulator: whatever evaluates that model from the current state. Manager: whatever starts the service and reconnects the stream. Services: exactly two – an alert-and-monitoring-centric service that raises the work order, and a state-centric one, because somebody on shore can see the residual.

Layers. A physical interface with one adapter that receives the stream. A core with the model, the state and a shadowing function that computes the residual. A digital interface with one adapter that reaches the maintenance system. No digital interface for anyone else: no other twin reads this one, no query answers questions about last month, no forecast runs on request.

What the architecture predicts. This "twin" is a state-centric and alert-centric service pair with a private digital interface. That is not a criticism; it is the honest shape of a shadow built for one decision. It also says what the next increment costs: a forecast service needs the model behind a slot and a simulator that can run it forward, which the alert never needed; and a second consumer needs a public digital interface, which is a new adapter, not a new twin. The blocks were all there; two of them were never exposed.

4.10.2 Example 2, one step left to you: the hospital ward occupancy twin

A hospital ward's twin: beds, admissions, expected discharges. The physical adapters read an admissions system, not a sensor – an event arrives when a patient is admitted, transferred or discharged. The state is the set of occupied and free beds and the expected occupancy over the next shift. A function-centric service answers "if we admit three more, where do they go tonight?" and a state-centric one shows the ward board.

Place the data, the functions, the model and the services yourself; they follow the pattern. Two of the seven need thought, and they are yours:

- **(a)** Where does the *manager* go, and what does it start and stop, when the physical side is a hospital information system rather than a device? What does "restart on failure" mean here?
- **(b)** What does *bound* mean for this twin? A sensor is bound when a channel to it is open. An admissions system is always reachable and always answering. Propose a definition of bound and shadowed that a ward manager would accept, and say what the twin should display when it is neither.

Hint for (b): the Layered Model section said state is meaningful only in one phase, and gave the reason. Ask what would have to be true for the ward board to be believed.

4.10.3 Example 3, posed as a problem: two hundred pumps

A water utility runs two hundred pumps across forty stations. Design the architecture of their twins, and in particular decide between one twin with two hundred physical adapters and two hundred twins under one manager. Use the interface-controller lesson of the Lessons from Five Reference Models section and the class-and-instance idea of the Digital Twin Configuration section. Say what the state of a station's twin is, whether it is composed from its pumps' or aggregates them, and what a pump's configuration would contain that the class's does not. Then say which of the eight principles your design would violate first if the utility asked for the twins to start writing speed setpoints.

4.11 Summary

Back to the objectives this chapter opened with.

1. **A twin is a software system of a particular kind.** Long-running and stateful, event-driven, a control system whose controller is software, with models as first-class artefacts. Its architecture is judged on the ordinary attributes – performance efficiency, maintainability, compatibility – and on one of its own: how faithfully and quickly its state tracks the physical twin.
2. **Seven kinds of thing live inside it.** Data, state, functions, models, simulators, manager, services. State is the centre: small, current, structured, with a core part bound to the physical twin that no application may write and a derived part that may be recomputed at will. Data is not state.
3. **Services sort into four families by what they read.** Data-centric, function-centric, state-centric, alert-and-monitoring-centric. The dashboard is a state-centric service, and the sentence is a compliment. The twin must be monitored as software, separately from the asset, because a lost connection looks like a quiet day.
4. **Eight principles, two patterns.** Two interfaces, never one; state as the single source of truth; the model behind a slot; events in and out; reusable assets; observability; an explicit lifecycle; a secured physical-digital boundary. Layered and service-oriented are the patterns most twins use, and the three-layer model is a layered pattern.
5. **The picture to remember has three layers.** Physical interface, core, digital interface; a shadowing pipeline of normalise, shadow, augment, consolidate, deliver; and a lifecycle in which state means something only when shadowed.
6. **Five reference models agree on four things and disagree on vocabulary.** A physical thing, a digital representation, a connection, data. They diverge on whether services, lifecycle and the interfaces are named parts. The seven blocks are what they name between them, not a sixth model.
7. **A configuration is the architecture written down, in three rungs.** A schema describes state; relationships describe how states refer to each other; a full configuration describes all seven blocks and where they run, and belongs in version control because a running twin that no longer matches it is a fault. On the rig, the whole architecture fits in a page, and Stage 3 adds one line to it.

The sentence to carry forward: **a digital twin is a long-running, event-driven software system with a state at its centre, a physical interface on one side and a digital interface on the other; everything else is a service attached to that state.**

4.12 Exercises

Solutions or hints follow the exercises; each says which. The objective each exercise serves is given in brackets. None of the exercises uses the demonstrator.

Exercise 1 (Objective 2, easy). Classify each service into one of the four families, in one line each: (a) an endpoint returning a turbine's current power output; (b) a nightly export of a compressor's readings to a spreadsheet; (c) a request that returns the optimal start-up sequence for a boiler; (d) a page that lights red when a tank level exceeds a threshold; (e) a page that lights red when the tank's twin has not received a reading for five minutes.

Exercise 2 (Objective 1, easy). A colleague describes a system: "pressure readings from the pipeline come in over MQTT into a Kafka topic, a job aggregates them into a Postgres table every minute, a Grafana board shows the table, and a Python script fits a leak model every night." Place each named part on the seven blocks, and name the block that is missing.

Exercise 3 (Objective 3, medium). Redraw the system of Exercise 2 – a polling script that writes a database – as the three-layer model. Say what it lacks at each layer, and which single addition would give it a state.

Exercise 4 (Objective 1, medium). Write the state document, in JSON, for a lift in an office building: current floor, direction, door state, load, the last command sent, and whatever derived fields you think a maintenance service would want. Mark which fields are core and which are derived, and justify one of each.

Exercise 5 (Objective 3, medium). A twin loses its connection to its physical twin. Write, as a numbered list, what the manager must do in the first second, the first minute and the first hour, and what the digital interface must serve at each of those moments.

Exercise 6 (Objective 1, medium). Argue for or against letting a service re-derive a quantity from the recorded data instead of reading it from the state. Give one situation in which re-deriving is the right call, and say what the state document would have to carry for the two answers never to disagree.

Exercise 7 (Objective 4, medium). Take one of the five reference models and, without looking at the comparison table, map its vocabulary onto the seven blocks. Then check against the table and explain any disagreement – yours or the table's.

Exercise 8 (Objective 5, medium). For a warehouse's conveyor line described as "twelve belt sections, each with a speed sensor and a motor controller, feeding two sorters", write a Thing-style schema for one belt section and a full configuration for the line, in the shape of the two listings in this chapter. Say which sections of the configuration would differ between two lines in two warehouses.

Exercise 9 (Objective 5, hard). Do Example 3 of the Faded Examples section in full, including the last question.

Exercise 10 (Objectives 3 and 4, hard, open-ended). The three-layer model came from a framework that was measured; the five-dimension model is a diagram. Construct one architectural question about a twin you are designing that the diagram cannot answer and the framework can, and one that the diagram answers and the framework does not. Say which kind of question your project has more of.

4.12.1 Solutions and hints

Exercise 1 (solution). (a) State-centric. (b) Data-centric. (c) Function-centric. (d) Alert-and-monitoring-centric, pointed at the asset. (e) Alert-and-monitoring-centric, pointed at the twin – the inward half, and the one that tells you whether (a) and (d) can be believed.

Exercise 2 (solution). MQTT and the Kafka topic: data, incoming, and the physical adapter that receives it. The aggregating job: a function. The Postgres table: data, recorded. The Grafana

board: a state-centric service – except that there is no state, so it is reading data and calling it state. The nightly fit: a model, and the script that runs it is a simulator of sorts. Missing: *state*, and with it a manager that knows whether the pipeline is bound and shadowing. The board shows the last row of a table, which is data, not a belief with an age and a phase.

Exercise 3 (hint). The physical interface is the MQTT subscriber; the core is the aggregating job and the model, with no state between them; the digital interface is Grafana's query against the table, which is a consumer reaching past the core into the data. It lacks a state document, a shadowing function that writes it, a lifecycle field, and a digital adapter of its own. The single addition: a state document the job writes and Grafana reads, with `as_of` and a lifecycle field on it. Everything else follows from having somewhere to put those two fields.

Exercise 4 (hint). Core: floor, direction, door state, load, last command – all reported by the lift or sent to it, none writable by a service. Derived: trips today, door cycles since service, an estimate of brake wear, a health score. Justify a core field by asking what would happen if a service could write it: a dashboard that could set "door closed" would be lying to the person about to step in. Justify a derived field by asking what happens if it is wrong: a wrong wear estimate costs a visit, not a life.

Exercise 5 (hint). First second: mark the lifecycle out of sync, stop advancing `as_of`, keep the last state. First minute: retry the connection with a bound number of attempts, raise the inward alert, serve the last state *with* the phase word on it. First hour: escalate the alert, stop serving derived fields whose staleness exceeds their horizon, keep serving core fields with their age. The digital interface never serves a number without its `as_of`, and never serves a phase other than the true one.

Exercise 6 (hint). Against, by default: two derivations drift, and the one a person looks at is the one that lies. For, in one case: a data-centric service answering "what was the average last week" is supposed to read the recorded data, because that question is about the record and not about now. For the two never to disagree, the state document would have to carry the recorded value it was derived from and the time it was derived, so that a re-derivation can be checked against it rather than argued with.

Exercise 7 (hint). The rows where you and the table are most likely to disagree are State and Manager, because those are the rows where the conceptual models have dashes. If you put "virtual entity" against State, ask whether a set of models is a current belief with an age. If you put "connection" against Manager, ask whether a connection knows whether it is bound.

Exercise 8 (hint). The schema for one belt section has attributes (section number, motor model) and features (speed, motor status, load). The line's full configuration has twelve instances of one class under `assets`, one `compute` section, a `dt_services` section with a state-centric line board and an alert on any section stopped, and a `physical_twin` section with twelve controller addresses and, if the line's twin is allowed to act, twelve write endpoints. Between two warehouses the class is the same and the instance addresses, the calibration and the identity differ – which is the point of the class.

Exercise 9 (open). Assessed on whether the choice between one twin and two hundred is argued from the interface-controller lesson and the class-and-instance idea rather than from taste; on whether the station twin's state is explicitly composed or aggregated; and on whether the last question names a specific principle – most good answers name the secured boundary or the explicit lifecycle – and says why that one goes first.

Exercise 10 (open). A question the framework answers and the diagram cannot: "how much latency does the shadowing pipeline add per event on this edge node?" A question the diagram answers and the framework does not: "have we accounted for services as a dimension of the design, or only as things we will build later?" Most software projects have more of the first kind; most project proposals have more of the second.

4.13 Further Reading

- *The five reference models, first-hand:* [123] for the five-dimension model and [124] for the software platform architecture the same group built on it; [87] for the analysts' walk through ISO 23247's entities and sub-entities, with [125] for the standard applied to use cases and [38] for how well published manufacturing architectures align with it; [126], [127], [128] and [129] for the Digital Twin Consortium's platform stack, its summary, its glossary and its systems-of-systems interoperability framework; [82] for WLDT; [85] for the six-dimension model and its interface controller.
- *Architecture studies:* [59] for the systematic mapping of digital-twin software architectures, their quality attributes and their patterns – the source of the "layered and service-oriented" finding; [56] for the pattern catalogue; [19] for the Industrial Internet Reference Architecture's "digital twin core as a middleware" pattern; [60] for the unifying reference model that maps four vocabularies onto common concepts.
- *The WLDT and Web of Digital Twins line of work:* [115] for the Web of Digital Twins, its model of properties, relationships and events, its core and augmented state and its lifecycle; [83] for the edge digital twin and its three parts; [112] for the five-step shadowing pipeline and the microservice and serverless realisations of it; [122] for the entanglement-aware lifecycle and the middleware that manages many twins; [81] for the twin's software lifecycle set against the physical twin's phases; [113] for the three layers from a security viewpoint and metric-based observability; [84] for hierarchical composition and relationship navigability; [135] and [130] for ecosystems of twins across platforms.
- *Configuration:* [119] for the open-source frameworks survey and the Eclipse Ditto workflow, with [132] and [55] for a framework built on the Ditto Thing; [133] for the comparison of digital twin description formats; [134] for DTDL's twin graph; [114] for the Digital Twin as a Service chapter – the configuration template, the minimal configuration, classes and instances, the lifecycle manager – with [46] for the case studies that fill the template in; [136] for GitOps applied to the twin lifecycle.
- *The reusable-asset and platform view:* [114] again, for assets and composition; [79] for the twin as an assembly of services built up from models and data, and [30] for the advanced services – what-if, fault diagnosis, predictive maintenance, reconfiguration; [116] for a DT Manager administering FMU models; [117] and [118] for co-simulation and its standard interface; [121] for the twin-as-a-service platform survey and its view of monitoring twin instances; [49] for a survey that places WLDT and Ditto among the platforms.
- *Consulted but not drawn on:* [131] for the Asset Administration Shell as a hierarchy of submodels, which belongs to the standards chapter; [77] for how data reaches a twin, which belongs to the data chapters; [53] for the greenhouse exemplar's architecture, cited once above as the rig's closest relative; [34] for the evolution of twins and their models over time, which the credibility chapter needs more than this one does.

Part II

The Building Blocks

Chapter 5

Data for Digital Twins

5.1 Before You Start

What you are assumed to know. You write software for a living, you are comfortable with HTTP APIs, JSON, a time-series or relational database, and reading Python you did not write, and you have read the previous four chapters. From Chapter 2 you have the plant rig's four low-level protocols – the I2C chain of sensors behind a multiplexer, the MODBUS bus that reads registers and writes coils, the InfluxDB point of measurement, tags, fields and timestamp, and the REST API on port 8099 whose sensing endpoints take a `since_timestamp` parameter. From Chapter 3 you have the two-question test and the staleness row of the shadow-versus-twin table: a shadow shows an old number, a twin acts on one and needs a bound. From Chapter 4 you have the seven building blocks, in particular the **data** block and the sentence that data is not state; the **physical adapter** that speaks the hardware's protocols; the shadowing pipeline whose first step is *normalise*; and the `as_of` field that says how old the state's belief is. Each is recalled in a clause where it is used.

What you are not assumed to know. Signal processing beyond arithmetic: the one theorem this chapter needs is stated in a single line, and the mathematics appendix at the end of the book has the argument. Any streaming, serialisation or time-series product: a few are named, only where a published system used them, and none is recommended. Any industrial standard: two protocols are introduced for what they do, and the standards chapter takes up the rest.

What this chapter covers. Chapter 4 named data as one of seven blocks and said that data is not state. This chapter is about that block: how a number gets from a sensor into the store the twin reads, and what is lost, gained and decided on the way. It reads the twinning loop as data and names the tuple the physical adapter produces; it sorts twin data three ways; it derives a sampling interval from a decision and shows what goes wrong when the interval is chosen from the datasheet instead; it reads a sensor's datasheet for the six numbers that change a design; it puts four timestamps on one path and two clocks behind them; it asks five things of any transport, names the protocols and shows one stream of readings as each carries it; it encodes one reading six ways and counts the bytes; it sorts a twin's functions by what they do and builds the ingestion pipeline as five of them; it places stream processing on a continuum from edge to cloud; it lays the same readings out as rows, columns, a time-series point, an object and a metadata record, and sizes the volume; it gives quality its own field; it ends with a choosing table, the rig's own data path, and three faded examples.

What this chapter deliberately does not cover. The downward leg of the loop – commands to the physical twin and the command path that guards them – beyond noting where it joins. State estimation, and every check that needs the model to say what a reading *should* have been, including the stuck sensor: those belong to the state and estimator chapter. How a model is built or kept current: the models chapter. Whether the twin deserves to be believed:

the credibility chapter, which will inherit this chapter's limitations paragraph.

Learning objectives. By the end of this chapter you will be able to:

1. **Read** a sensor's datasheet for the six numbers that change a design, and **decide** whether a measurement can support a stated decision.
2. **Derive** a sampling interval from a decision, and **design** the aliasing check that tests it.
3. **Evaluate** any transport against the five properties a twin needs, and **place** HTTP, Message Queuing Telemetry Transport (MQTT), Open Platform Communications Unified Architecture (OPC UA), MODBUS and Kafka against them.
4. **Carry** event time and ingest time through a pipeline, **size** clock error against the sampling interval, and **compute** twinning rate and age of twin.
5. **Choose** an encoding for a value in flight and a store and layout for data at rest, and **say** what each preserves and loses.
6. **Sort** a twin's functions by what they do, **write** the functions of an ingestion pipeline as testable, versioned units, and **place** each on the adapter or the core.
7. **Design** a time-series schema, a retention policy, a cold-path layout and a metadata record, and **size** the volume before it is a problem.

Every section that follows serves at least one of those seven, and the exercises at the end say which.

5.2 The Twinning Loop, Read as Data

Chapter 3 drew the twin as a control loop: readings go up from the physical twin, commands come down to it. Chapter 4 opened the digital object and put a state at its centre. This chapter is the upward leg, from the sensor to the store, and it ends where Chapter 4's state begins. Figure 5.1 draws the loop with the upward leg labelled by the stations this chapter visits.

Take an offshore wind turbine as the generic instance for this section and most of the chapter. Its nacelle carries a few hundred sensors – bearing temperatures, generator speed, tower accelerations, wind speed and direction – wired into a controller that the turbine's maker designed for the turbine's own control, not for you. A twin of that turbine begins where a reading leaves that controller.

The physical adapter of Chapter 4 speaks whatever the hardware speaks. What it produces, whatever it spoke, is one thing: a **tuple** of six fields – binding, quantity, value, unit, event time and quality – and the Transports section fixes that tuple as the adapter's output contract once the transports that feed it have been named. Until then, hold one sentence: the adapter's whole output is a stream of tuples, and every section of this chapter is about one of the six fields or about the stream they form.

5.2.1 Why the adapter is a quarter of the engineering

In a project plan the adapter is one line, estimated as "read the API", and in the delivered system it is a quarter of the engineering. Three reasons, each of which the rest of the chapter unpacks.

The physical twin was not designed for you. The turbine's controller exposes what its maker needed for control, in the units, rates and naming its maker chose, over the protocol its maker's suppliers used. A register holding the bearing temperature in tenths of a degree, as a sixteen-bit integer, big-endian, refreshed every second, is a normal thing to be handed. Every one of those facts has to be discovered, written down as a binding, and applied exactly once. The middleware literature treats this heterogeneity as the problem middleware exists to absorb: twins, like the Internet of Things (IoT) before them, encompass heterogeneous devices, software

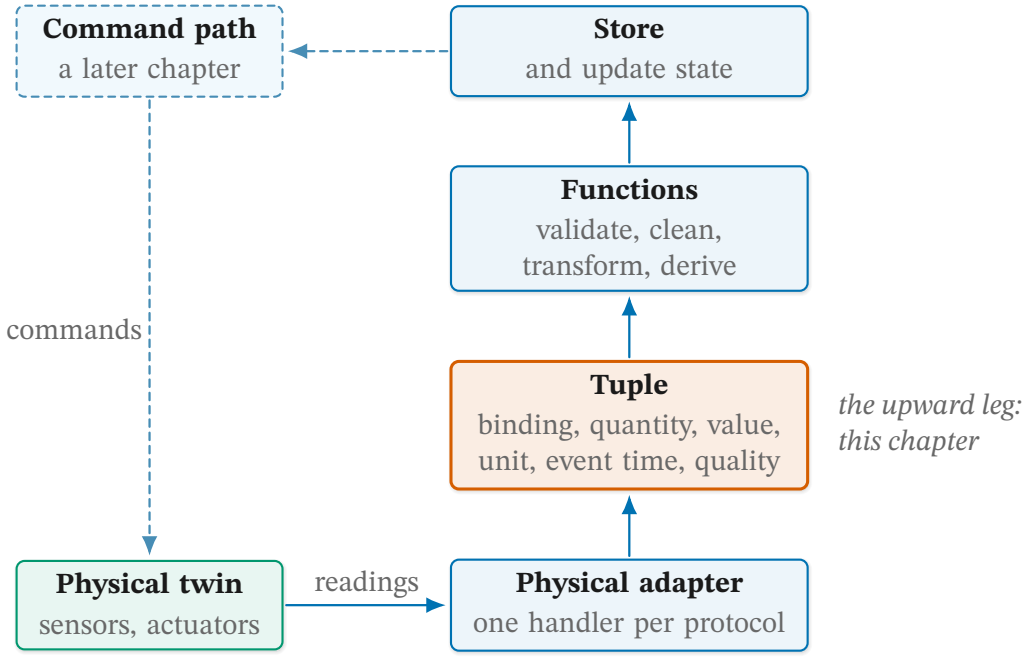


Figure 5.1: The twinning loop read as data. Readings climb the right-hand side from the physical twin through the adapter's tuple and the functions into the store and the state; commands come back down the left through a command path a later chapter owns. This chapter is the upward leg.

components, communication protocols and legacy applications, and middleware's traditional job is to hide that behind standardised interfaces [137]. In a twin that job lands on the adapter.

Every failure in the physical world arrives through this component. A sensor fails, a cable is cut, a gateway reboots with the wrong time, a firmware update changes a scale factor. None of those events announces itself. Each arrives at the adapter as a tuple that looks wrong, or as the absence of a tuple that was due. The adapter is the only component positioned to notice, and the Quality section is about giving it the vocabulary to say what it noticed.

Every number the credibility argument will rest on is a tuple it produced. When the credibility chapter asks whether the twin's model is good enough to act on, the evidence is a comparison between what the model predicted and what the sensors reported. If the sensors' readings were mis-scaled, mis-timed or silently filled in, the comparison is meaningless and nothing downstream can tell. A large part of whether a twin deserves to be believed is settled in the adapter, before any model runs.

5.3 A Vocabulary for Twin Data

Chapter 4 sorted the data block into incoming, recorded and configuration. That sorting is by where data sits. This section adds two more – by where it came from and by how quickly it must be used – and a third that runs across an asset's life. Table 5.1 puts the three side by side.

5.3.1 By origin

Sensed data came from a sensor through the adapter: a tuple with a binding to something physical. **Simulated** data came from a model run by a simulator: the same shape of tuple, with a binding to a model version rather than to a sensor. **Derived** data was computed by a function

from sensed or simulated data: a residual, a rolling mean, an efficiency. **Configuration** data is what the twin was told about itself and its asset: bindings, calibration constants, thresholds, firmware versions.

Derived data is the category people forget to store, on the argument that it can be recomputed. It cannot, once the function that derived it has changed. A residual computed by last year's threshold is a different number from one computed by this year's, and if only the sensed data was kept, the record of what the twin believed at the time – and therefore of what it did, and why – is gone. Store derived data, and store beside it the version of the function that made it. The Functions section returns to the versioning; here the point is the storing.

5.3.2 By timeliness

Assign every service to one of three paths by how quickly it needs its data. The **hot path** is for answers within seconds or less – an alarm on a vibration threshold, a command-path check. The **warm path** is for answers within minutes – a residual recomputed per check interval, a dashboard that refreshes. The **cold path** is for answers within hours to months – a fatigue summary, a recalibration, a report. The names come from the data-analytics reference architectures that the manufacturing-twin literature reviews, one of which defines three paths for data after ingestion: a hot path for near-real-time insight through stream analytics, a warm path for more detailed processing at larger delay, and a cold path for batch processing at longer intervals [138].

Assigning each service to a path fixes three other decisions at once – the protocol that carries its data, where it runs, and which store it reads – which is why the path is the first thing to decide and the last to change. A hot-path alarm cannot wait for a nightly file; a cold-path report should not be reading a message broker. The How to Choose section is built on this observation, and the table there is organised by path.

5.3.3 By lifecycle

An asset has a life – design, manufacture, operation, retirement – and data about it is produced in every phase by different tools that never meet. The **digital thread** is the name for the linkage of one asset's data across those phases. In the Digital Twin Consortium's definition it is the means by which information about an asset is tied together along several axes of its virtual representation – time and lifecycle stage among them, with the design intent counted as a stage [129]; an accessible overview puts the same idea as a twin being both a shadow of its physical twin's present state and a thread recording its evolution over time, which applications can scan and draw meaning from [17]. For a software engineer the thread is a habit more than a technology: record, on every row the twin stores, which firmware produced the reading and which model version produced the prediction, so that a number from March can still be interpreted in November after both have changed.

Table 5.1: Three sortings of twin data. Chapter 4's incoming, recorded and configuration sorting is by where data sits; these three are by where it came from, how fast it is needed, and which phase of the asset's life produced it.

Sorting	Classes	What it decides	Turbine instance
By origin	sensed, simulated, derived, configuration	what must be stored, and with which version	a bearing temperature; a predicted temperature; their residual; the bearing's alarm threshold
By timeliness	hot, warm, cold	protocol, placement, store	vibration alarm; per-check residual; monthly fatigue summary
By lifecycle	design, manufacture, operation, retirement	which firmware and model version a row was produced under	the blade's design loads; its factory inspection; its operating history; its decommissioning record

Read Table 5.1 across its last column: one bearing temperature appears in all three rows, as a sensed reading, as a hot-path alarm input and as a row stamped with a firmware version, and the three sortings are three questions asked of the same tuple rather than three kinds of data. None of the three is state: the Functions section says what turns data into state, and why nothing in this chapter does.

5.4 Sampling: From Physical Quantity to Number

A sensor reports a continuous quantity; the twin stores a sequence of numbers. Between the two sit two decisions that are usually made by default and should be made on purpose: how often to read, and how finely to round. This section makes both, and then shows the failure that the default reading interval produces.

5.4.1 Five steps from a decision to an interval

The sampling interval is derived from the decision, not chosen from the datasheet. Five steps:

1. **Name the decision and how long it may take.** For the turbine: "take the turbine out of service before a bearing fails", and the decision may take a day, because a vessel has to be scheduled.
2. **Name the physical event that matters to the decision, and how long it takes to become visible in the quantity you measure.** A bearing that is failing runs hot over days to weeks before it fails; its temperature climbs over hours.
3. **Sample several times across that event.** If the temperature climbs over hours, a reading every ten minutes sees the climb as a curve rather than as one surprising number. Several means at least a handful; the appendix says how many, and why.
4. **Check the ceiling.** The sensor's own response time and maximum sample rate, the bus that carries the reading, and the power budget of a battery-fed gateway all set a rate above which more readings cost more and tell no more.

5. **Check the floor.** The decision's latency sets a rate below which the twin learns too late. A day's decision latency with a ten-minute reading is comfortable; a ten-minute reading against a decision that must be made within a minute is not.

Write the interval as T_s, and the rate as $f_s = 1/T_s$. For the bearing, $T_s = 600$ seconds. Now change the decision. "Shut the turbine down within a second if the tower's acceleration exceeds its design load" names a physical event – a gust, a resonance – that becomes visible in milliseconds, and step three then asks for readings a few milliseconds apart: T_s in the hundreds of microseconds, f_s in the kilohertz. Same turbine, two quantities, two decisions, intervals four orders of magnitude apart. The interval belongs to the decision, and a twin that reads everything at one rate has made one of the two decisions wrong.

5.4.2 Aliasing: fast behaviour appearing as slow behaviour

The failure that step three guards against has a name. **Aliasing** is fast behaviour appearing as slow behaviour when the quantity is sampled more slowly than it changes. The record then contains a rhythm that does not exist, and nothing in the record says so.

One worked instance. A building's ventilation fan cycles on and off on a timer, one full cycle every 594 seconds – a "ten-minute" cycle that is in fact nine minutes and fifty-four seconds. The room's temperature follows it with a small ripple. The building twin samples the room temperature every 600 seconds, because ten minutes seemed reasonable for a room. Each sample lands six seconds later in the fan's cycle than the one before, so the samples walk slowly round the cycle, and the record shows a smooth, slow oscillation whose period is the time it takes six seconds per sample to add up to one full cycle:

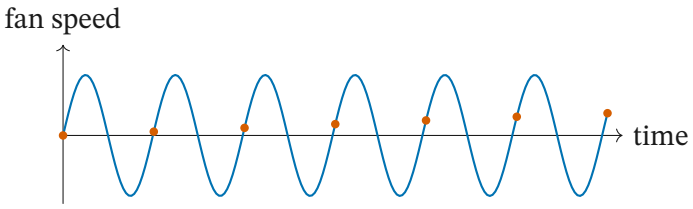
$$f_{\text{alias}} = \frac{1}{594} - \frac{1}{600} = \frac{6}{356{,}400} = \frac{1}{59{,}400}$$

per second, a period of 59,400 seconds – sixteen and a half hours. Figure 5.2 draws it. Panel (a) is one hour of the true signal with the six samples marked on it; panel (b) is a day of the samples alone, a clean phantom rhythm that a colleague will explain as "the building warming through the afternoon and cooling at night" and a model will be fitted to.

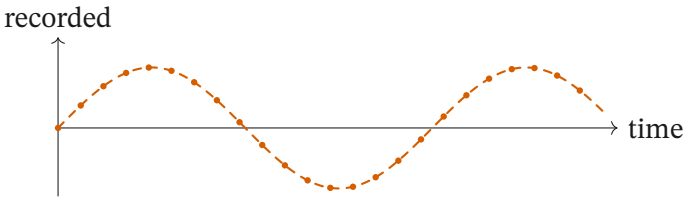
The empirical check costs an afternoon and is worth it for every quantity a twin samples. Sample the quantity much faster than you intend to, once – every second, for a day. Then **decimate**: keep every six-hundredth reading, which is what your intended interval would have seen. Compare the decimated record with the fast one. If the decimated record contains a rhythm the fast one does not, you have found an alias, and the fix is to sample faster, or to average the fast readings over each interval rather than pick one, or to move the sampling instant so that it does not beat against a known cycle. The rule the check is testing is the one theorem this chapter states, and it fits on a line: a sampled record represents the quantity faithfully only when the sampling rate exceeds twice the fastest rate at which the quantity changes, $f_s > 2f_{\max}$. The mathematics appendix has the argument and the reason "several times across the event" in step three is the practical form of it.

5.4.3 Quantisation: the ten-bit calculation

The second decision is how finely to round. An analogue-to-digital converter reports one of a finite set of counts, and the **quantisation step** q is the size of one count in the quantity's own unit. Work it for a ten-bit converter reading a temperature sensor that outputs ten millivolts per degree, powered from a 3.3 V reference. Ten bits give $2^{10} = 1024$ counts across the reference, so one count is



(a) a 594 s cycle sampled every 600 s, one hour



(b) the same samples over a day: a phantom 16.5 h rhythm

Figure 5.2: Aliasing. In (a) a room temperature that follows a fan's 594-second cycle is sampled every 600 seconds; in (b) the samples alone, over a day, draw a slow wave with a period of sixteen and a half hours that the room never had. Nothing in the record of (b) says it is a phantom.

$$q = \frac{3.3}{1024} = 0.00322$$

volts. At ten millivolts per degree that is $0.00322/0.010 = 0.322$ degrees per count. The converter rounds to the nearest count, so the worst-case rounding error is half a count:

$$e_q = q/2 = 0.16 \tag{5.1}$$

degrees. Equation 5.1 is the floor under every later use of this reading: a residual of a tenth of a degree from this sensor is inside the rounding and means nothing, and a compression that promises to reconstruct the value within a tenth of a degree is promising more precision than the reading ever had. The Storage section uses that second fact. Note also which number the calculation did not use: the sensor's accuracy, which the Sensors section defines. Quantisation is the converter's error; accuracy is the sensor's; the two add, and a reading's honest uncertainty is never smaller than the larger of them.

5.4.4 Sampled versus event-driven acquisition

Everything so far has been **sampled acquisition**: read the quantity at a fixed interval, whether or not it changed. Its alternative is **event-driven acquisition**: emit a tuple when something happens – a breaker trips, a door opens, a dose is delivered, a value crosses a threshold. Events carry information that sampling misses (the breaker tripped at 06:10:14, not "sometime in the ten minutes before 06:20") and they carry nothing when nothing happens, which is efficient and dangerous. Silence from an event-driven source means either that nothing happened or that the source is dead, and the two are indistinguishable from the record. The resolution is a **heartbeat**: a periodic tuple that says "nothing happened, and I am alive". With it, silence has a meaning; without it, a dead sensor and a quiet day are the same record.

The sentence to carry: sample the state, capture the events, align them on time. Quantities that describe a condition – temperatures, pressures, levels – are sampled. Things that happen – trips, doses, commands, alarms – are captured as events. And both are placed on one timeline

by their event time, which is the Time section's subject. First, though, the sensor that produced the sample, and what its maker actually promised about it.

5.5 Sensors: What You Are Actually Given

A sensor is a model of the world that somebody else built and documented. A capacitive soil probe measures capacitance; the datasheet claims a relationship between that capacitance and the water content of the soil around it, and the claim holds under the conditions the datasheet states and nowhere else. A bearing's temperature sensor measures the temperature of a small metal bead pressed against the bearing housing, and the datasheet claims a relationship between that bead and the bearing. Everything the twin will ever know about the physical twin arrives through such claims, and reading them is the skill this section teaches.

5.5.1 Six numbers

A datasheet is long. Six numbers on it change a design, and Table 5.2 names what each one changes. Two of them, resolution and maximum sample rate, the Sampling section has already leaned on; the other four are new here.

Table 5.2: The six datasheet numbers that change a design, and what each changes. Range and resolution are about what can be reported; accuracy and drift about how far to trust it; response time and sample rate about how fast it can be asked.

Number	What it says	What it changes downstream
Range	the smallest and largest values the sensor reports	the out-of-range quality state; whether a fault can be seen at all
Resolution	the smallest change the sensor can report	the quantisation step; how many digits to store; the floor on any threshold
Accuracy	how far a report may be from the truth, under stated conditions	how small a residual can mean anything; the error bound a compression may use
Drift	how accuracy degrades with time or temperature	the recalibration interval; whether a slow trend is the asset or the sensor
Response time	how long the report takes to follow a change in the quantity	the ceiling on useful sampling rate; the lag the model must expect
Maximum sample rate	how often the sensor can be read	the ceiling on the sampling interval; the bus load

A published fermentation twin shows what the numbers look like when they are written down: its sampling chamber's sensor array has an optical dissolved-oxygen sensor with a range of 0 to 100 mg/L and an accuracy of plus or minus 0.05 mg/L, and an industrial pH sensor with a range of 0 to 14 and an accuracy of plus or minus 0.002 pH with temperature compensation, beside conductivity, temperature and pressure sensors and an electronic hydrometer for density [139]. Read those two accuracies against the decisions they serve. A brewer deciding whether a fermentation has stalled needs to see a change of a few tenths of a pH unit over hours; an accuracy of two thousandths is a hundred times better than the decision needs, and the design

question becomes whether the sensor's drift over a three-week fermentation stays inside that margin. The same paper notes that the manual practice it replaces was a master brewer drawing samples by hand for gravity, dissolved oxygen, pH and conductivity – which is the decision the sensors were bought to improve, and the number to read the datasheet against.

Calibration is the number the datasheet cannot give, because it is a date rather than a figure, and it is the cost most often skipped. It does not stay skipped: a twin fed with real-time sensor data is fed with data affected by environmental factors, wear and calibration issues, which show up as data loss, noise and inconsistencies [140]. A sensor whose calibration has drifted reports a trend that the asset does not have, and the Quality section's `uncertain` state is the only honest thing the adapter can attach to it until somebody recalibrates.

5.5.2 Resolution is not accuracy

Resolution and accuracy are the most commonly confused pair on the table, and the confusion runs one way. Resolution is how finely a sensor can *report*; accuracy is how far the report may be from the *truth*. A reading of 21.437 degrees from a sensor accurate to half a degree has four digits, two of which are decoration. Nothing in the tuple says so. The value arrives with three decimal places because the analogue-to- digital converter had that many bits, and a dashboard shows them because they are there, and a colleague reads "21.437" as a claim about the world when the honest claim is "somewhere between 20.9 and 21.9".

Two consequences for the twin. Store the value at the sensor's resolution and no finer – extra digits cost storage and invite false precision. And never set a threshold, an alarm or a residual test finer than the accuracy: a rule that fires when the residual exceeds a tenth of a degree, on a sensor accurate to half a degree, fires on the sensor and not on the asset.

5.6 Time: Four Timestamps and Two Clocks

A tuple carries an event time. Where that time came from, how it is written, and how far it may be trusted decide whether two readings can be compared at all. This section sits before the Transports section because the five properties a twin asks of a transport presuppose that event time exists and is preserved.

5.6.1 Four stations on one path

Follow one bearing temperature from the sensor to the twin's store, and count the places a clock is read. Figure 5.3 draws them.

The **sensor read** is when the quantity was measured, by the sensor's own clock if it has one, or by the gateway's when it polled. The **gateway record** is when the gateway wrote the reading into its local log or message queue; between the first and the second lie the bus and the gateway's firmware. The **transport** stamps are when the message was sent and when it was received by a broker or server; between the second and the third lie a queue, a radio and a network. The **ingest time** is when the twin's adapter turned the message into a tuple and wrote it; between the third and the fourth lie the broker and the twin's own load. Each gap has a different cause, and each has a different owner: the sensor's maker, the gateway's integrator, the network's operator, the twin's developer. When a reading arrives late, the first question is which gap grew, and a path that recorded all four stamps can answer it. A path that recorded only the last cannot.

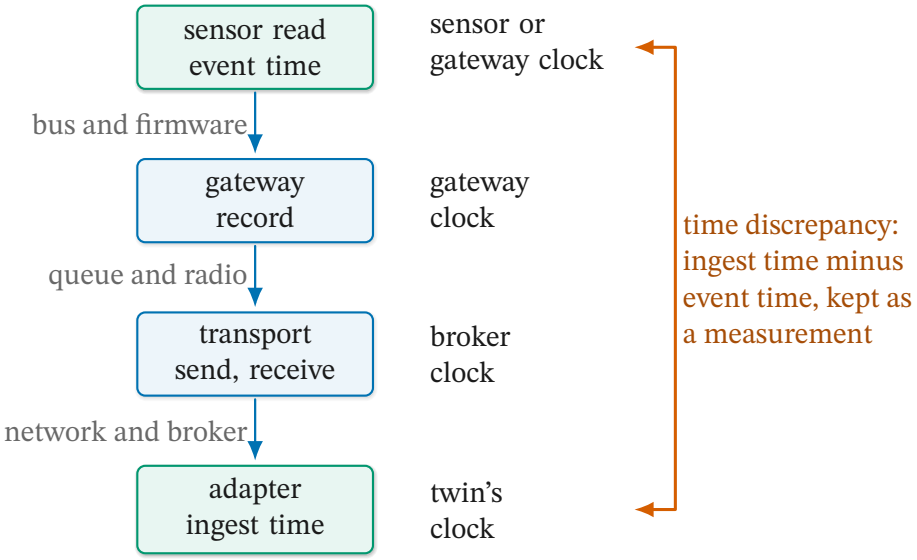


Figure 5.3: Four stations on one reading's path, each with a clock and each gap with a different cause and owner. The tuple keeps the first and the adapter records the last; their difference is the time discrepancy.

5.6.2 Event time, ingest time, and the difference between them

Of the four, two must survive into the tuple and the store. The rule: the tuple carries the **event time** t_e, the adapter records the **ingest time** t_i, and their difference is not an error to be hidden but a measurement to be kept, because it is the one number that says how far behind the physical twin the digital one is running. Write it as

$$D = t_i - t_e \tag{5.2}$$

and call it the **time discrepancy**. Equation 5.2 has been studied in its own right for twins: the coupling between a twin and its physical counterpart is subject to time delays and synchronisation errors, timeliness can be critical when a twin must warn a physical system of an obstacle or when delay threatens a control loop's stability, and one line of work brokers external data into a running co-simulation in real time precisely to manage that delay [141]. It is also an incident indicator: a study that subjected a twin to denial of service, packet loss with jitter, replay, out-of-order delivery and node failure found that each left measurable deviations in the twin's operational metrics – timeliness, reliability, availability and consistency – and proposes metric-based observability as a lightweight way to detect that something is wrong [113]. A time discrepancy that grows is the first of those metrics to move, and it costs one subtraction.

The failure the rule prevents is the adapter stamping each tuple with the time it *arrived* and calling that the event time. On a quiet network the two are seconds apart and nobody notices. When the gateway buffers an hour of readings through a link outage and replays them, every reading in that hour carries the same minute, the record shows the bearing's temperature leaping and freezing, and the estimator of Chapter 7 fits a model to the network outage.

5.6.3 The four timestamp questions

Every timestamp a twin stores should be able to answer four questions.

Whose clock? The sensor's, the gateway's, the broker's or the twin's. Two timestamps from two clocks can be compared only if the clocks agree, and the next subsection is about how much they must.

What offset? A time written without its zone is a time written in somebody's local convention. Store every timestamp in Coordinated Universal Time (UTC), and let the dashboard convert. A twin that stores local time loses an hour every autumn and gains one every spring, and the gap and the duplicate it produces are indistinguishable from a sensor fault.

What resolution? A timestamp to the second cannot order two readings taken a millisecond apart. Store event times at the resolution the sampling interval needs – milliseconds for a kilohertz stream, seconds for a ten-minute one – and no coarser.

Start or end of interval? A ten-minute average stamped 06:10 either describes 06:00 to 06:10 or 06:10 to 06:20, and the two conventions are both in use. Decide once, write it in the binding, and never let a downstream function guess.

5.6.4 Clock error as a fraction of the sampling interval

Clock error matters as a fraction of the sampling interval. Write the error between two clocks as ε and the sampling interval as T_s; the quantity to look at is ε/T_s. Work it twice, with opposite answers.

A gateway clock one second wrong, against a bearing temperature read every ten minutes: $\varepsilon/T_s = 1/600$, under two tenths of a per cent of the interval. The reading is placed a second from where it belongs on a curve that changes over hours. Noise. Network time synchronisation, which holds a clock within tens of milliseconds over an ordinary network, is more than this reading needs.

The same one second, against a tower acceleration sampled at two kilohertz: $T_s = 0.0005$ seconds, so $\varepsilon/T_s = 2000$. Two thousand samples are in the wrong place. An analysis that compares two accelerometers on the tower to find the direction of a vibration is comparing samples a second apart and calling them simultaneous, and the direction it finds is fiction. This stream needs a clock disciplined to microseconds – precision time protocols over a wired network, or a satellite time reference at the gateway – and the decision to buy that is made here, from this ratio, not from the sensor's datasheet.

One special case of clock error is not a fraction at all. A gateway that reboots without a battery-backed clock starts at some epoch – the year 1970, or the firmware's build date – and stamps every reading with an impossible time until it synchronises. Those tuples must be rejected, not corrected: a reading stamped 1970 cannot be placed, and guessing where it belongs is inventing data. The validate function of the Functions section owns that rejection, and the Quality section gives it a state to attach.

5.6.5 Twinning rate, age of twin, time discrepancy

Three numbers, computed per binding, say how well the upward leg is working. The **twinning rate** R is how often the twin's state is updated from the physical twin – distinct from the sensor's sampling rate, which may be higher, with the gateway averaging, or lower, with the adapter interpolating. The manufacturing literature defines it as the synchronisation rate between physical and virtual entity and notes that most published twins did not distinguish it from the data sampling frequency [91]; the platform literature normalises it to an interval where zero means no synchronisation and one means real-time, and observes that a low rate leads to an outdated state [121]. Compute it as updates per unit time for each binding, $R = n/T$, and alarm when it falls.

The **age of twin** A is the time that has elapsed since the newest reading the state rests on:

$$A = t_{\text{now}} - t_{\text{e}} \tag{5.3}$$

where t_e is the event time of that newest reading. The platform survey defines age of twin as the time elapsed since the twin last received an update from its physical counterpart, the temporal freshness of the twin [121]; Equation 5.3 uses event time rather than arrival time so that a replayed backlog does not make a stale twin look fresh. This is the best single health number a twin has. It says, in one figure, whether the twin is shadowing at all (A grows without bound when the link is down), whether it is keeping up (A stays near T_s when it is), and whether a command may be issued (Chapter 3's staleness bound is a bound on A). It is what Chapter 4's `as_of` field exists to make computable: `as_of` is t_e for the state, and A is now minus that. Every state-centric service should show it, and the lifecycle field should fall to *out of sync* when it exceeds a bound the configuration names.

The **time discrepancy** D, from Equation 5.2, is the third, and the one whose growth points at a gap on the path rather than at the asset. Together the three answer "is the twin current?", "how current?" and "if not, whose fault?" – and all three are computable from the tuple's event time and the adapter's clock, which is the whole argument for carrying both.

5.7 Moving the Data: What a Twin Needs from a Transport

A reading exists first as a voltage, then as a register, then as a message, then as a tuple. This section is about the middle two steps: how a value travels, and what a twin must be able to ask of whatever carries it.

5.7.1 Two layers and a gateway

Transports come in two layers, and Figure 5.4 puts a gateway between them.

The **device layer** is the buses Chapter 2 introduced and a few more of their kind: the Inter-Integrated Circuit bus (I2C) and the Serial Peripheral Interface (SPI) over centimetres on a board; MODBUS over RS-485 and the Controller Area Network (CAN) over tens of metres in a cabinet or a vehicle; the 4–20 mA current loop, in which a sensor signals a value as a current between four and twenty milliamps down a two-wire pair, so that zero current means a broken wire rather than a zero reading. These carry volts, registers and coils. They have no names for quantities, no units and no timestamps; what they carry is a number at an address.

The **system layer** is where the twin's software lives: messages over a network, each with a name, a unit if the sender attached one, and a time. HTTP, MQTT, OPC UA and a durable log such as Kafka live here.

Between them sits the **gateway**: Chapter 2's Raspberry Pi, a programmable logic controller (PLC) in a cabinet, an industrial edge box. It polls the device layer, turns a register into a number, attaches a name and a time, and emits a message. A gateway is where a register becomes a number and a number becomes a message; the twin's adapter is where a message becomes a tuple. Which of those two does the scaling and the unit is a decision, and the section's last subsection makes it.

5.7.2 Poll versus subscribe, and the question that outranks both

Every system-layer transport is either **polled** – the twin asks, the source answers – or **subscribed** – the twin registers an interest and the source, or a broker on its behalf, pushes. Table 5.3 contrasts them.

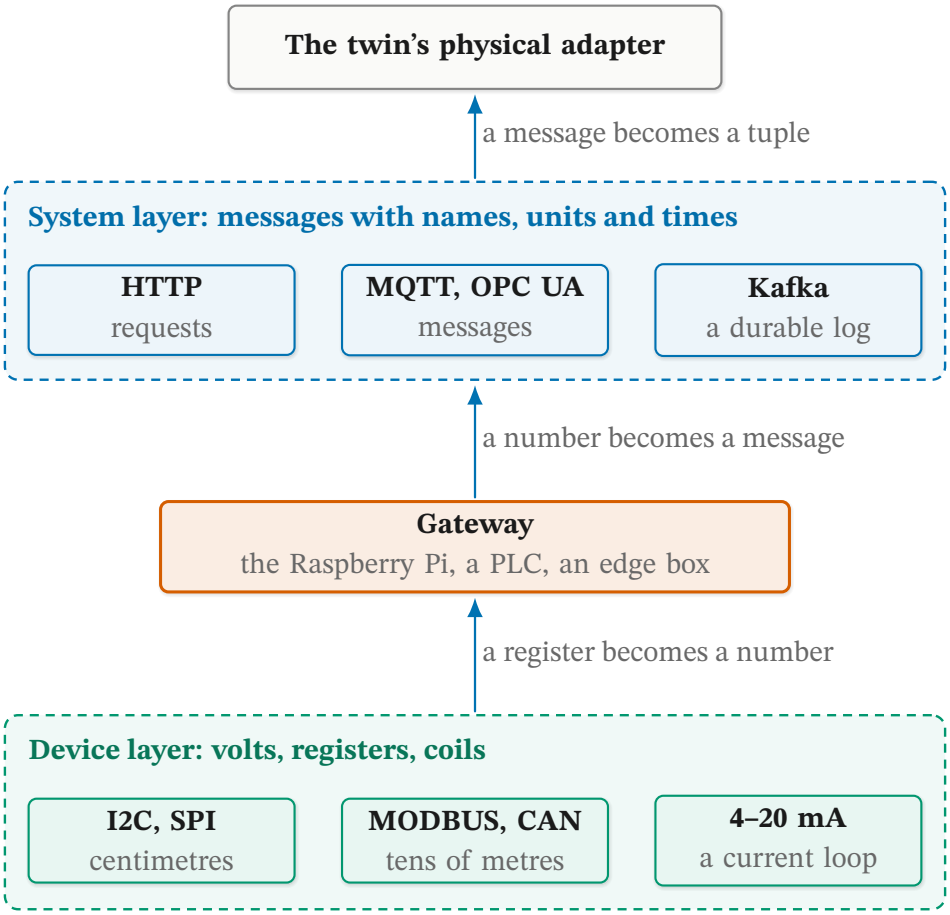


Figure 5.4: Two layers of transport. The device layer carries volts and registers over short buses; the system layer carries messages with names, units and times; the gateway is where one becomes the other, and the twin's adapter reads the system layer.

Table 5.3: Poll versus subscribe. Neither is better; the row that decides between them for a twin is the last one, and the question below outranks both.

	Poll	Subscribe
Who initiates	the twin	the source or a broker
Timing of a reading	when the twin asks	when the source publishes
Cost when nothing changes	a request and an empty answer	nothing
Cost when much changes	readings missed between polls	a burst the twin must absorb
Connection state	known at every request	known only through keep-alives or a broker's notice
Fan-out to many consumers	each polls the source	the broker fans out

	Poll	Subscribe
What a gap looks like	a poll that failed, visible to the twin	messages that never arrived, invisible unless numbered

Resumability – **can I ask you for what I missed** – is the question that outranks the choice between the two. A transport that can answer it turns a dropped connection into a delay; one that cannot turns it into a gap that nothing downstream can fill. A polled source with a history parameter – give me everything since this time – is resumable. A subscribed feed whose broker keeps a log and lets a subscriber resume from where it left off is resumable. A subscribed feed whose broker delivers each message once, to whoever is connected, and forgets it, is not, however low its latency. Ask this question of every transport before asking any other.

5.7.3 Five properties a twin needs

Chapter 3's obligations arrive at the transport as five properties, each of which has a victim when it is missing. Table 5.4 names them.

Table 5.4: Five properties a twin needs from any transport, and the component that breaks when each is missing. The five are the transport-side form of Chapter 3's staleness, fail-safe and concurrency obligations.

Property	The question	Who breaks without it
Event time preserved	does the message carry when the reading was taken, not when it was sent?	the estimator, which fits a model to arrival times
Connection state observable	can the twin tell a live source that is quiet from a dead one?	the monitoring service, for which a lost link looks like a quiet day
Gap survivable	after an outage, can the twin recover what it missed?	the record, which has holes nobody can see
Ordering knowable	can the twin tell whether two messages arrived in the order they happened?	the shadowing function, when a correction arrives before the reading it corrects
Deduplication possible	if a message is delivered twice, can the twin tell?	every aggregate, which counts a retried reading twice

Score a transport by counting the properties it gives you for free, then by counting the ones you can add above it – a sequence number in the payload adds ordering and deduplication to anything – and reject the ones where a property cannot be added at all. The scores are worked for the rig in the demonstrator section and for a turbine in the faded examples.

5.7.4 Protocols

With the questions settled, the protocols. One paragraph each on what a protocol is for, in the order a reading meets them from the device upward, and beside each a sample of the same data as that protocol carries it; the How to Choose section ranks them by path, not here. The data is a stream of bearing temperatures from the turbine, read every ten minutes, written here as JSON

with its timestamps – the stream the Encodings section will re-encode and the Storage section will store:

```
{"binding": "wt07/gen-bearing/temp", "event_time": "2026-06-09T06:00:00Z",
 "value": 61.1, "unit": "degC", "quality": "good"}
{"binding": "wt07/gen-bearing/temp", "event_time": "2026-06-09T06:10:00Z",
 "value": 61.2, "unit": "degC", "quality": "good"}
{"binding": "wt07/gen-bearing/temp", "event_time": "2026-06-09T06:20:00Z",
 "value": 61.4, "unit": "degC", "quality": "good"}
```

One object per reading, shown here over two lines each, with the tuple's fields as keys; the quantity is folded into the binding's name, which is a convention the binding records. Each protocol below carries these readings differently, and what differs between them is exactly the five properties of Table 5.4.

HTTP is for polling, and Chapter 2 has already used it. It is a request/response protocol that addresses resources by a Uniform Resource Identifier (URI), text-based, with no fixed header or payload size [80]. Its resumability is entirely in the API's hands: a sensing endpoint that accepts a history parameter – readings since a given time – makes polling resumable and gives the twin ordering and deduplication through the timestamps it returns; one that returns only the latest value gives none of the five. The turbine's controller, if it offers such an endpoint, is asked for the stream like this:

```
GET /turbines/wt07/sensing/gen-bearing/temp?since=2026-06-09T05:50:00Z
Host: scada.windfarm.example
Accept: application/json
```

and answers with a JSON array of the readings since that time – the three objects above, inside `[` and `]`, newest first or oldest first as the API documents. The `since` parameter is the whole of the resumability; without it the same endpoint returns one object and the twin learns nothing about what it missed. HTTP's cost is the request itself, paid whether or not anything changed, and its virtue is that every request reveals the connection state.

MQTT is for brokered telemetry. It is a publish/subscribe transport standardised by the Organization for the Advancement of Structured Information Standards (OASIS) and by ISO, designed to minimise network bandwidth and device resource requirements: a publisher sends a message on a named topic, a broker relays it to every subscriber of that topic, and the fixed header is two bytes [80, 142]. Brokers differ in the features they support – three quality-of-service levels, a retain flag that keeps the last message on a topic for late subscribers, persistent sessions, and a last will and testament published by the broker when a client disappears [142]. One reading, published:

```
topic:    fleet/wt07/gen-bearing/temp
qos:      1         retain: false
payload:  {"seq": 4127, "event_time": "2026-06-09T06:10:00Z",
           "value": 61.2, "unit": "degC", "quality": "good"}
```

The binding has moved into the topic name, and one field has appeared that the stream did not have: `seq`. Bare MQTT gives a twin little of the five – no event time unless the payload carries one, connection state only through the will message, no gap recovery, no ordering or deduplication across a reconnect – and the industrial conventions layered on it add those back in the payload: a sequence number so a gap and a duplicate are visible, a timestamp so event time survives, a will message registered at connection so the broker announces a client that vanished. Read against the table: connection state, ordering and deduplication, added by convention on

top of a transport that had none of them, and every one of those conventions is the publisher's promise rather than the protocol's.

MODBUS is the register-based poll/response protocol of legacy PLCs, recalled from Chapter 2 and belonging to the device layer. A security survey characterises it in a line: a legacy industrial protocol that works on request/reply messages and has no inherent encryption [143]; the predictive-maintenance review finds it among the protocols the twins it surveyed used to reach equipment [25]. The same reading, as the gateway polls it from the turbine's controller – the request and reply with the bus's own address and checksum bytes left off:

```
request:  03  00 2A  00 01     read holding registers, address 42, count 1
reply:    03  02  02 64        function 3, two bytes follow: 0x0264 = 612
```

Two bytes carry the value, and nothing carries anything else. That 612 is tenths of a degree Celsius, that register 42 is the generator-side bearing of turbine 7, that the reply arrived at 06:10:00 by the gateway's clock and that the reading is believed good – every field of the tuple but the value – exists only in the binding and in the gateway that applies it. So every one of the five properties has to be supplied by whoever polls it, and that is the gateway's job: poll, stamp with its own clock, name the register from the binding, and publish upstream with a sequence number.

OPC UA is the protocol that carries an information model as well as values. It is based on two communication models, client/server and publish/subscribe, and provides a semantically enriched information model to represent data and its meaning [144]; a client browses a server's address space to discover variables and the information attached to them [145]; generic device models can be instantiated for particular devices, and both a publish/subscribe pattern and a request/response pattern are supported on the same address space [91]. It standardises manufacturer-independent data exchange between physical objects in industrial environments [25] and stands as the reference standard for machine interoperability: the interoperability literature treats it as one of the main standards that other descriptions are mapped onto [144]. The same reading as an OPC UA data value, in the shape of its JSON encoding and simplified:

```
{"NodeId": "ns=2;s=WT07.GenBearing.Temp",
 "Value": {"Type": "Double", "Body": 61.2},
 "SourceTimestamp": "2026-06-09T06:10:00Z",
 "ServerTimestamp": "2026-06-09T06:10:00.412Z",
 "StatusCode": "Good"}
```

Two timestamps, not one: the source's and the server's, which are the Time section's sensor read and gateway record carried by the protocol itself rather than by a convention. A status code that maps onto the tuple's quality. And no unit in the message, because the engineering unit is a property of the node in the address space, read once when the binding is discovered. For the twin that address space is the difference from MODBUS: a reading arrives with its name, its unit and its timestamp attached, so event time is preserved and the binding can be discovered rather than typed.

Kafka is a durable, replayable log behind the gateway, and not a device protocol. In the publish/subscribe evaluation that compares it with brokers, the broker does the heavy lifting; Kafka's subscribers read partitions in order, each message carrying an offset that marks its position, so a subscriber that disconnects and reconnects continues from the last message it read, and many consumer groups can read the same topic at once [146]. The same reading as a record in the log:

```
topic:      fleet.readings     partition: 7     offset: 88213417
```

```
key:        wt07/gen-bearing/temp
timestamp:  2026-06-09T06:10:00.655Z        (the broker's, when appended)
value:      {"binding": "wt07/gen-bearing/temp",
             "event_time": "2026-06-09T06:10:00Z",
             "value": 61.2, "unit": "degC", "quality": "good"}
```

The payload is the stream's own line, unchanged; what the log adds is outside it. The offset is the log's position, not the publisher's count, so a consumer that resumes from offset 88213416 receives this record again and can tell that it has. That is resumability, ordering and deduplication as properties of the log rather than as conventions in the payload, which is why a durable log sits so often between the adapter and everything else: the data-analytics reference architecture uses Kafka topics as its ingest stream, camera images published to one topic and classification results to another [138]. It is the Processing section's durable buffer.

5.7.5 One handler per protocol, one form upstream

Whatever the transport, the shape of the code is the same: one **handler** per protocol, one normalised form upstream. The handler is where register numbers, multiplexer channels and scaling factors die, and it is the only place a unit is attached. Everything above it sees the tuple. An edge-twin framework shows the shape at scale: devices speaking MQTT, HTTP and CoAP with their own JSON layouts are each shadowed by an instance that homogenises the protocol to MQTT and the data format to one standard, and the overhead of doing so was measured at ten messages a second with hundred-byte payloads and found comparable to the framework's other configurations [83]. Two rules follow. A scale factor is applied in exactly one handler, never in a handler and again in a function, which is the failure the Encodings section works. And the handler is tested with a captured frame and an expected tuple, which is the only test that catches a byte-order error before the turbine's bearing reads minus forty degrees.

5.7.6 The adapter's output contract

What every handler produces, whatever protocol it spoke, is one thing: the **tuple** of six fields the Twinning Loop section named, now fixed as the adapter's output contract.

```
(binding, quantity, value, unit, event_time, quality)
```

The **binding** names which sensor or channel produced the reading – this turbine, this bearing, this register – and is part of the twin's configuration. The **quantity** says what was measured. The **value** is the number. The **unit** is attached exactly once, here. The **event time** says when the physical event happened, by the best clock available where it happened. The **quality** says how far the value can be trusted, in one of six states the Quality section defines. Chapter 4's shadowing pipeline began with a step called *normalise*; this tuple is what normalise produces, and everything above the adapter – the functions, the shadowing function, the state, every service – reads tuples and nothing else. That is what lets the rest of the twin be written once for every protocol. Look back at the five samples in the Protocols subsection: the MODBUS reply carried two bytes, the OPC UA data value carried two timestamps and a status code, the Kafka record carried an offset, and the handler for each turns what it was given into the same six fields. A bearing temperature that arrived as a MODBUS register and one that arrived as an MQTT message are the same tuple by the time anything else sees them, and the code that reads them does not know which was which.

The adapter's whole output is a stream of tuples, and every section of this chapter is about one of the six fields or about the stream they form: the Sampling and Sensors sections were about

the value, the Time section about the event time, this section and the Encodings section about how the stream travels, the Functions and Processing sections about what is computed from it, the Storage section about where it lands, and the Quality section about the sixth field.

5.8 Encodings: A Value in Flight

A tuple's value spends its journey encoded, and each encoding is a set of conventions about how a number is laid out in bytes or characters. This section takes one reading from the Transports section's stream – the bearing at 61.2 degrees at 06:10:00 – and encodes it six ways, from the register up, then organises the encodings by one distinction: whether a reader with nothing but the message can tell what is in it.

5.8.1 A register is a convention, applied once

At the device layer a value is a register: a sixteen-bit integer with a scale factor, an offset and a byte order, and every one of those is a convention that must be applied exactly once. The bearing temperature of 61.2 degrees stored as tenths of a degree is the integer 612; stored as hundredths with an offset of minus forty degrees, it is 10,120; stored big-endian it arrives as the bytes `02 64`, little-endian as `64 02`. Applied twice, the scale turns 61.2 into 6.12; applied zero times it stays 612 and looks like a temperature, a hot one; applied with the wrong byte order it becomes 25,602 and looks like a fault. None of these failures is visible in the tuple, because the tuple carries a number and a unit and both look fine. The binding records all three conventions, the handler applies them, and no other code touches them. Two bytes on the wire is the first row of Table 5.5, and the cheapest encoding there is, because everything but the value is somewhere else.

5.8.2 Text and binary

At the system layer the value is inside a message, and the message is in one of two families. **Text encodings** are readable by a person and verbose on the wire. The reading as JSON is 124 bytes when written on one line:

```
{"binding": "wt07/gen-bearing/temp", "event_time": "2026-06-09T06:10:00Z",
 "value": 61.2, "unit": "degC", "quality": "good"}
```

and as XML, with the fields as attributes, 116:

```
<reading binding="wt07/gen-bearing/temp" event_time="2026-06-09T06:10:00Z"
         unit="degC" quality="good">61.2</reading>
```

Neither needs anything but itself to be read: the field names are in the message, the types are visible from the syntax, and a person can read either in a browser. **Binary encodings** are compact, and how compact depends on what they leave out. The Concise Binary Object Representation (CBOR) is JSON's data model in bytes – the same reading is 101 bytes, and here is how they are spent:

```
A5                                  a map of five pairs
  67 62 69 6E 64 69 6E 67           "binding"                 (7 bytes)
  75 77 74 30 37 2F 67 ... 6D 70    "wt07/gen-bearing/temp"   (21 bytes)
  6A 65 76 65 6E 74 5F 74 69 6D 65  "event_time"              (10 bytes)
  74 32 30 32 36 2D 30 ... 30 5A    "2026-06-09T06:10:00Z"    (20 bytes)
  65 76 61 6C 75 65                 "value"
```

```
  FB 40 4E 99 99 99 99 99 9A         61.2 as a 64-bit float
  64 75 6E 69 74   64 64 65 67 43    "unit" "degC"
  67 71 75 61 6C 69 74 79            "quality"
  64 67 6F 6F 64                     "good"
```

CBOR saved 23 bytes on JSON, and the arithmetic says why so few: the five field names are still in the message, the timestamp is still twenty characters of text, and the one number became nine bytes where JSON spent four characters. A binary encoding that keeps the names is still self-describing, and still nearly as large. The saving comes from dropping the names, and dropping the names means both ends must agree on a schema. **Protocol Buffers** does that with numbered fields. The schema:

```
message Reading {
  string  binding    = 1;
  int64   event_time = 2;   // seconds since 1970-01-01T00:00:00Z
  float   value      = 3;
  string  unit       = 4;
  Quality quality    = 5;   // an enum: good = 1, uncertain = 2, ...
}
```

and the reading, 42 bytes:

```
0A 15 77 74 30 37 2F ... 6D 70   field 1, 21 bytes: the binding
10 B8 DC 9E D1 06                field 2: 1780985400, a varint
1D CD CC 74 42                   field 3: 61.2 as a 32-bit float
22 04 64 65 67 43                field 4: "degC"
28 01                            field 5: good
```

Each field is a one-byte tag and a value; the names are in the `.proto` file, not on the wire, and the timestamp is five bytes of integer instead of twenty of text. Replace the binding string with an integer id that the twin's configuration maps back to a name, let the unit be implied by the binding, and the same reading is 16 bytes. **Avro** goes one step further and drops the tags too. Its schema is itself written in JSON:

```
{"type": "record", "name": "Reading", "fields": [
  {"name": "binding",    "type": "string"},
  {"name": "event_time", "type": "long"},
  {"name": "value",      "type": "float"},
  {"name": "unit",       "type": "string"},
  {"name": "quality",    "type": {"type": "enum", "name": "Quality",
     "symbols": ["good", "uncertain", "substituted",
                 "out_of_range", "no_reading", "not_attempted"]}}]}
```

and the reading is 37 bytes: the binding's length and characters, the timestamp as a variable-length integer, four bytes of float, the unit's length and characters, and one byte for `good`, in the schema's order and nothing between them. A reader without that exact schema cannot even find where one field ends and the next begins.

One number in those examples deserves a second look. Protocol Buffers and Avro carried 61.2 as a 32-bit float, which holds 61.2000008 – a rounding of its own, of about a ten-millionth of a degree. By the Sampling section's rule that is free: the sensor's resolution is a tenth of a degree, and a rounding a million times finer than the sensor's loses nothing the sensor had. A 64-bit float, as CBOR used, buys four more bytes of precision the reading never had.

Table 5.5 puts the six side by side.

Table 5.5: One reading of the bearing temperature in six encodings. The bytes fall as what the reader must already know grows; the two-byte register at the top is the cheapest encoding and the one that carries least.

Encoding	Bytes for one reading	What the bytes carry	What the reader must already know
Raw register	2	one integer	the binding: scale, offset, byte order, unit, when it was read
JSON	124	names, values, types by syntax	nothing
XML	116	names and values, as attributes	nothing; a schema may be referenced
CBOR	101	names and typed values	nothing
Protocol Buffers	42, or 16 with an integer binding id	numbered fields and values	the `.proto` file
Avro	37	values in schema order	the writer's exact schema

Read Table 5.5 down its last column and the saving in the second is bought with the fourth: every byte taken off the wire is a fact moved into a schema somebody must keep. The platform survey lists the family a twin will meet – JSON, CBOR, CSV, XML, Protocol Buffers and YAML among the structured formats – and states why the choice matters: a protocol carries data in an encoding largely dictated by the protocol, the encoding affects the size of the payload and hence the communication latency, and compact formats such as CBOR and Protocol Buffers reduce that overhead [121]. On a constrained link – a radio, a metered cellular connection, a satellite backhaul from a turbine at sea – payload size is latency, and the choice of encoding is a choice about how far behind the twin runs. Three readings an hour at 124 bytes is nothing on any link; two thousand tower accelerations a second at 124 bytes each is two megabits a second before a single reading has been stored, and at 16 bytes it is a quarter of that.

One protocol offers both families of the same message. OPC UA's publish/subscribe part complements its client/server model with a binary and a JSON encoding, and chooses its transport by direction: for lateral connectivity on a local network it uses the User Datagram Protocol (UDP), where a missing packet causes no wait for a correction, which suits time-sensitive networks and constrained devices; for vertical connectivity to cloud environments it uses MQTT or the Advanced Message Queuing Protocol (AMQP) [147]. One information model, two encodings, two transports, chosen by where the message is going – which is the choice this section is about, made inside one standard.

5.8.3 Self-describing or schema-required

The distinction that organises every encoding: an encoding is either **self-describing**, so that a reader with no other information can recover the field names and types from the message, or **schema-required**, so that the schema must live somewhere the reader can reach. Table 5.6 sorts the encodings worked above.

Table 5.6: Encodings sorted by whether the message describes itself. The second is smaller on the wire and the first is smaller in the organisation: a self-describing message costs bytes, a schema-required one costs a place to keep the schema and a rule for changing it.

Encoding	Family	Self-describing?	Where the schema lives
A raw register	binary	no	in the binding: scale, offset, byte order
JSON	text	yes, at the level of names and types	nowhere required; a schema may be published beside the API
XML	text	yes, and the schema may travel with it	in a schema document the message may reference
Protocol Buffers	binary	no	in a `.proto` file both ends compile from
CBOR	binary	yes, at the level of types	as for JSON
Avro	binary	no, per message	in a JSON schema, sent once or kept in a registry

The second family is smaller on the wire and the first is smaller in the organisation, and the choice is about which of the two is scarce. A turbine at sea has scarce bandwidth and a fixed set of quantities: a schema-required encoding, with the schema versioned in the twin's configuration, is the right answer. A greenhouse on a local network with sensors that change every month has scarce engineering time: a self-describing JSON message that a person can read in a browser is the right answer, and the extra bytes cost nothing. When the schema is not in the message it lives in one of three places – compiled into both ends, sent once when a connection opens, or kept in a registry both ends can reach – and each of those is a thing to version, because a schema that changes under a stored message makes the message unreadable. The Avro reading above is the sharpest case: 37 bytes that mean a bearing temperature under one schema and nothing at all under another.

5.8.4 Units as fields or as convention

Units travel either as a field in the message or as a convention between the people who wrote the two ends, and only the first can be checked by a program. A JSON reading with `"unit": "degC"` beside its value can be validated; one whose unit is understood because the sender's documentation says so cannot, and the documentation is the first thing to go stale. A unit that travels as a field should be a code from a list rather than a free string: the AutomationML practical guide recommends the United Nations Economic Commission for Europe (UNECE) Recommendation 20 codes for units of measure, using each unit's common code rather than the unit's own symbol, so that a program can compare two units by comparing two codes [148]. The handler attaches the unit once; a code list is what lets the validate function check it.

5.9 Functions: The Computations Between Reading and State

Chapter 4 defined functions as stateless, testable, reusable across twins, and said they were the software engineer's home turf. This section first sorts the functions a twin has by what they

do, because only some of them belong to this chapter, and then builds the ones the upward leg needs.

5.9.1 Shadowing, augmentation, and the other things a function does

A twin's functions are named by what they do for the twin, and the survey the platform literature most often builds on lists what a twin does beyond holding a model: it *reflects* its physical object, representing its properties, conditions and behaviours through models and data and remaining its virtual counterpart across the object's whole lifecycle [149]. That reflection needs a link, which the same survey calls *entanglement*: the exchange of status information between the physical object and its logical counterpart, in real or near-real time, with connectivity as its first property and a direction that is typically physical to logical but valuable in both [149]. Around those two it names the rest: *memorisation*, so that the twin can be represented and controlled over a large part or the whole of the life cycle; *composability*, the grouping of several objects into a composed one whose behaviour is observed and controlled as a whole and by component; *augmentation*, which turns an object whose functionalities and services were fixed for its whole life into a programmable one; and *predictability*, which with memorisation lets the twin project its behaviour into the future [149].

The platform work that followed turned two of those into named functions. The **shadowing function** is the one Chapter 4 put in its pipeline: it subscribes to the events arriving from the physical side, processes them against one or more models, and computes and maintains the twin's state [150]; in the Web of Digital Twins the same process also runs downward, propagating an action requested on the twin to the physical asset by a further shadowing function, so that a twin may mirror a lamp's switch as well as its brightness [115]. **Augmentation** is the other: a twin is a software replica whose responsibility is to clone the physical asset's available resources and functionalities *and* to extend existing behaviours with new capabilities, and a design that treats twins as passive data structures, each without its own shadowing process, forgoes exactly the augmentation and composition capabilities the recent literature describes [83]. Table 5.7 sorts the functions a twin runs by those names, and says which of them this chapter builds.

Table 5.7: The functions a twin runs, sorted by what each does. This chapter builds the first and fourth rows and reads the second from a distance; the rest are named so that the reader can tell, when a later chapter builds one, which kind it is.

Function type	What it computes	Reads, and writes	Where in this book
Ingestion	valid, clean, transformed tuples from protocol frames	frames and the binding; writes the store	this chapter, below
Shadowing	the twin's belief about the asset, from tuples and a model	the store, the model, the last state; writes the state and `as_of`	Chapter 4's pipeline; the state and estimator chapter
Augmentation	what the asset cannot compute about itself: a residual, a forecast, an alarm	the state and the models; writes derived data and the services	the derive function here; the models and services chapters
Memorisation	the history, kept and summarised	the tuple stream; writes the stores	this chapter's Storage section

Function type	What it computes	Reads, and writes	Where in this book
Composition	a fleet's belief from its members'	other twins' states; writes a composed state	the manager and platforms chapters
Reflection downward	a command for the asset from a request on the twin	the state and the command path's checks; writes the physical interface	the manager chapter

Read Table 5.7 for what each row writes and the line this chapter draws is visible. Everything the first and fourth rows write is data: a five-year history of bearing temperatures is data, however it is stored. The twin's current belief about the bearing – its estimated temperature, the time that estimate describes, and whether the twin is currently shadowing the turbine at all – is state, and the second row is the only one that writes it. Chapter 4 said data is not state; this table says which function makes the difference. The ingestion functions built below are the part of shadowing that runs before any model is consulted, and nothing in this chapter writes the state document. It fills the store the shadowing function reads, and the state and estimator chapter is about how well that function reads it.

5.9.2 Five functions, five ways to fail

The ingestion pipeline is five functions, and they fail five different ways. Figure 5.5 draws them in order.

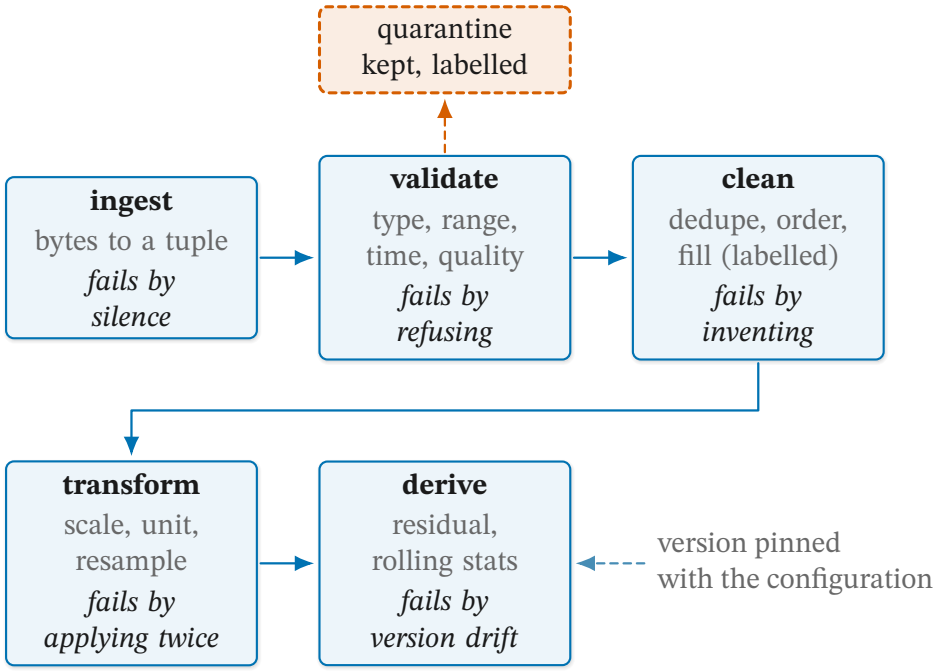


Figure 5.5: The ingestion pipeline as five functions. Each has a different failure: ingest fails by silence, validate by refusing, clean by inventing, transform by being applied twice, derive by drifting from the version that made yesterday's numbers. A refused tuple goes to quarantine, not to the bin.

Ingest turns bytes into a tuple: the handler of the Transports section. It fails by silence – a source that stops, a subscription that lapses – and its test is the heartbeat and the age of twin.

Validate checks the tuple against what the binding says it should be: the value is a number, inside the sensor's range; the event time is plausible, not 1970 and not tomorrow; the unit matches the binding's; the quality is one of the six states. It fails by refusing, and refusal is its job. Here it is, in the shape it takes for a temperature binding:

```
def validate(t, binding, now):
    if not isinstance(t.value, float):
        return refuse(t, "not a number")
    if not binding.range_lo <= t.value <= binding.range_hi:
        return refuse(t, "out of range")
    if t.event_time > now + binding.clock_slack:
        return refuse(t, "event time in the future")
    if t.event_time < binding.commissioned:
        return refuse(t, "event time before the sensor existed")
    if t.unit != binding.unit:
        return refuse(t, "unit mismatch")
    return accept(t)
```

Every threshold in it comes from the binding, none is a literal, and `refuse` writes the tuple to quarantine with the reason rather than dropping it.

Clean repairs what validate let through but the store cannot take: it removes duplicates, puts out-of-order tuples in event-time order, and fills a short gap if the binding says a gap may be filled. It fails by inventing – a filled value stored as if measured – and the Quality section's `substituted` state is the guard.

Transform applies a scale, converts a unit, resamples a stream to a common interval. It fails by being applied twice, which is the Encodings section's register turned into 6.12 degrees, and its guard is that a transform records what it did in the tuple's provenance so a second application can see the first.

Derive computes new quantities from clean ones: the residual of Chapter 2, a rolling mean, a rate of change, a spectral peak. It fails by drifting from the version that produced yesterday's numbers, and its guard is the versioning below.

Quarantine, do not discard. A burst of readings that fail validation is a diagnostic signal about the sensor, the link or the firmware – a gateway that rebooted to 1970 produces a thousand refusals in a row, and those thousand tuples, with their reasons, are the evidence of what happened. A pipeline that drops them has thrown away the evidence of its own fault and will report a gap it cannot explain.

5.9.3 Which run where

Chapter 4's table of blocks on layers put functions in two places, the adapter and the core, and the pipeline splits along that line. Decoding, scaling, unit attachment and validation run **in the adapter**, because they need the binding and nothing else, and because the tuple that leaves the adapter must already be a tuple. Residuals, rolling statistics and everything that needs the model or the state run **in the core**, because they read the state. The Industrial Internet Reference Architecture draws the same line from the edge side: the edge performs protocol adaptation, syntactic translation from binary readings to JSON and semantic translation such as unit conversion, and may run edge computing functions such as a fast Fourier transform on vibration data, while the information domain's functions ingest, check quality, augment with metadata, persist and distribute data as it arrives [19]. The placement is not fixed for all time: fluid digital twins relocate computational functions along the edge-to-cloud continuum as conditions change [150], and serverless platforms package a function as a unit the platform places and scales on its own

– function as a service, in the cloud-native survey's terms [151]. The Processing section takes up where; here the rule is what: a function that needs only the binding is the adapter's, a function that needs the state is the core's.

5.9.4 Testing and versioning

Test a function with a reading and a number. A validate function is given a tuple with a value of 612 degrees and a binding whose range ends at 150, and is expected to refuse; a transform is given the bytes `02 64`, a big-endian binding with a scale of a tenth, and is expected to return 61.2. There is no ceremony and no fixture beyond the binding, and that is the reason functions are the block where ordinary unit testing applies without argument.

Version them with the configuration, because a threshold is a decision somebody made. When the range in a binding changes, or the residual's formula changes, the derived data in the store changes meaning from that moment, and a row that does not say which version made it cannot be compared with a row that does. Every derived tuple carries the version of the function that made it; every function's parameters live in the configuration Chapter 4 put in version control; and the digital thread of the Vocabulary section is, for functions, exactly this habit.

5.9.5 The check no adapter function can make

One fault defeats every function in this section. A **stuck sensor** reports the same value, forever. The value is a number, inside the range, with a plausible time, in the right unit – valid, clean and well formed – and it is wrong, because the bearing's temperature is not holding to a hundredth of a degree through a day's operation. No adapter function can catch it, because catching it requires knowing what the reading *should* have done, and that is a question for the model. This chapter states the fault, gives it the `uncertain` quality state when a person notices, and hands the detection to the state and estimator chapter, where that estimator compares what arrived with what was predicted and can say that a value that never moves is the one value it did not expect.

5.10 Processing: Streams, Windows and Placement

The functions of the previous section run over the stream of tuples, and three decisions govern how: whether they run as the tuples arrive or over a batch later, over which span of tuples, and on which machine.

5.10.1 Stream or batch

A **stream** computation runs on each tuple, or each small window of them, as they arrive; a **batch** computation runs over a stored span later. The rule for choosing: if the answer changes the physical system's behaviour within that system's own time constant, the computation belongs on the stream; if not, a batch is cheaper and easier to test. The turbine's vibration alarm must fire within a second of the tower exceeding its load, because the turbine can be damaged in seconds: stream. Its daily fatigue summary changes a maintenance plan measured in months: batch, run at night over the day's Parquet files. The residual per check interval sits between: a stream computation on a warm path whose window is one check interval.

5.10.2 Windows

A window is the span of a stream over which one result is computed, and there are three shapes. Figure 5.6 draws them on one timeline.

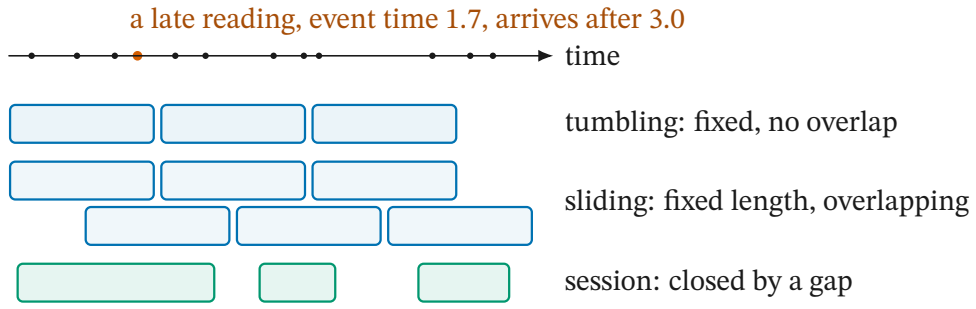


Figure 5.6: Three windows on one stream. Tumbling windows are fixed and do not overlap, so each reading is in exactly one; sliding windows are fixed and overlap, so each reading is in several; session windows are closed by a gap in the readings rather than by the clock. The late reading belongs to a window that has already closed.

Tumbling windows are fixed in length and do not overlap: the ten-minute mean of the bearing temperature, one result per ten minutes, each reading in exactly one. **Sliding** windows are fixed in length and overlap: a one-hour rolling mean recomputed every minute, each reading in sixty results. **Session** windows are bounded by a gap in activity rather than by the clock: one result per start-up of the turbine, from the first reading after a stop to the last before the next. Choose by the question – a per-interval summary tumbles, a trend slides, a per-episode statistic is a session – and note that the three windows are defined on **event time**, not arrival time, which is what makes the next problem a problem.

5.10.3 Late data

A tuple whose event time falls in a window that has already closed is **late**, and every stream system does one of three things with it. **Drop** it. Hold each window open for a **grace period** after its end and accept late tuples until then. Or **emit corrections**: close the window on time, and when a late tuple arrives, recompute and publish the window's result again with a note that it supersedes the first.

Drop is the silent default. A window that closed before a reading arrived, in a system that never said what it does with late readings, has dropped it, and the record shows nothing. The gateway that buffered an hour through an outage and replayed it delivers an hour of late tuples, and a pipeline built without a late-data policy discards the hour while reporting every window on time. The policy belongs in the configuration, per binding, because the right answer differs: a hot-path alarm drops, because a late alarm is no alarm; a warm-path residual uses a grace period of one check interval; a cold-path summary emits corrections, because a fatigue total that omits an hour is wrong for the rest of the turbine's life.

5.10.4 Edge, fog and cloud

Where a function runs is a placement decision on a continuum: on or beside the asset (the **edge**), on a site or regional node (the **fog**), or in a datacentre (the **cloud**). The placement literature treats it as an optimisation with measured constraints: one study places the components of a componentised application across edge, fog and cloud segments, deploys the result on a testbed, and derives guidelines under infrastructure constraints and response-time targets [152]; the twin middleware work envisions a twin as an ecosystem of microservices and serverless functions whose activation, mobility and replication along the continuum are driven by cyber-physical metrics, and observes that mature orchestration for that is still being sought [112]. There is measured evidence that offloading can win: a model of edge-only against edge-to-cloud completion

time, as a function of link speed, task complexity and edge processor load, shows conditions under which shipping a big-data job to the cloud finishes sooner than running it at the edge, and conditions under which it does not [153]. For the upward leg the rule of thumb follows the paths: hot-path functions run at the edge, because the round trip to the cloud is longer than the budget; cold-path functions run in the cloud, because the data is already there and the compute is cheap; and warm-path functions go wherever the bandwidth argument of the Storage section allows.

5.10.5 The durable buffer

Between the adapter and everything downstream sits, in most twins of any size, a durable log – the Kafka of the Transports section or something like it. Its job on the upward leg is to make a slow consumer harmless: because the log keeps every tuple and each consumer reads from its own offset, a consumer that falls behind or restarts continues from where it was and loses nothing [146], and because the adapter writes to the log rather than to the consumer, a slow consumer never applies back-pressure to a sensor. Without the buffer a slow dashboard slows the adapter, and a slowed adapter drops readings, and the record has a gap whose cause was a chart. With it, the gap becomes a lag, measured by the age of twin, and lags are recoverable where gaps are not.

State estimation – combining the stream with a model to produce the state Chapter 4 put at the centre – is the next computation the stream feeds, and it is the state and estimator chapter's subject. This section ends where the tuples are clean, windowed, placed and buffered; that chapter begins by reading them.

5.11 Storing Twin Data

One twin, several stores, and none of them owned by the twin: Chapter 4 said the store is a neighbour the twin reads and writes, and this section says how to lay it out. It carries the Transports section's bearing temperatures through every store they land in, so that the same three readings can be seen as rows, as columns, as a point in a time-series database, as an object in a bucket, and as the metadata record that says what they mean.

5.11.1 Several stores, none owned

A twin of the turbine has, at the least, a time-series database for the hot and warm paths, an object store for the cold path and for anything that is not a row, and a metadata store for what the readings mean. Each is read by other software than the twin and outlives any one version of it. The volume is in the first two, the meaning is in the third, and the shape the data takes in each – rows, columns, arrays – is the first thing to settle.

5.11.2 Rows and columns

Most telemetry is a table: one row per reading, columns for the tuple's fields. The three readings as a table are three rows under one header:

```
binding,event_time,value,unit,quality
wt07/gen-bearing/temp,2026-06-09T06:00:00Z,61.1,degC,good
wt07/gen-bearing/temp,2026-06-09T06:10:00Z,61.2,degC,good
wt07/gen-bearing/temp,2026-06-09T06:20:00Z,61.4,degC,good
```

That is a comma-separated values file (**CSV**), the universal export and the interface a supervisory control and data acquisition (SCADA) system will give you when nothing else is offered. It has no types, no units and no schema beyond its header row – nothing in the file says that `61.1` is a number and `degC` is not – and every one of those is supplied by the person who reads it. Treat a CSV export as a transport in disguise – resumable if the file is dated, ordered only if sorted – and land it in something better. JSON Lines, one JSON object per line, is the same idea with types and names on every row; the Transports section's stream was written in it, and it is what many pipelines write when they need an append-only text file.

The same three rows, stored by column instead:

```
event_time:  06:00:00Z   06:10:00Z   06:20:00Z   ...   144 values a day
value:       61.1        61.2        61.4        ...   neighbours alike
quality:     good        good        good        ...   one distinct value
binding:     wt07/gen-bearing/temp                     stored once
```

A row-oriented store keeps each reading's fields together, so writing one reading is one write and reading one reading is one read. A column-oriented store keeps each field's values together, so the mean of one field over a month touches one column and skips the rest. Telemetry is columnar at rest because the questions asked of it are columnar: nobody asks for "the reading at 06:10:00", everybody asks for "the bearing temperature over June", and a column of temperatures compresses far better than a row of mixed fields because its neighbours resemble each other – the value column above changes by a tenth of a degree per entry, the quality column does not change at all, and the binding is written once for the whole file. The hot path, which writes one reading at a time and reads the latest, is row-shaped; the cold path, which writes once and reads columns, is column-shaped; and a twin holds both.

Parquet is the columnar file the cold path lands in. Turbine owners generally store their sensor data as Parquet files in an object store and analyse it with Julia, Matlab, Python or R [154]; the ModelarDB system writes Parquet or Optimized Row Columnar (ORC) files to the local file system at the edge and to a distributed file system in the cloud [155]; and practitioners' first answer to compressing high-frequency turbine data is lossless big-data formats such as Parquet or ORC written at the edge and transferred [156]. **Arrow** is the in-memory column format that lets tools share data without copying. The same time-series system returns query results in Arrow so that programs can consume them directly, and transfers compressed segments between its edge and cloud instances over Arrow's own transport protocol rather than as rows [155].

5.11.3 Arrays and fields

Not everything is a row. Some data is a grid, an image or a cloud of points, and forcing it into rows destroys the structure a model needs.

Gridded environmental data – temperature over a sea surface, rainfall over a catchment – is a multidimensional array with named axes, and NetCDF is its file format: a marine twin of a coastal lagoon gathers its feeds from several providers, one of which delivers its datasets in NetCDF over HTTP [106]. **Labelled multidimensional arrays** are the working form for simulation inputs and outputs: a plant-modelling environment builds its inputs and outputs as labelled arrays and data sets so that index- and label-based selection and interpolation are available on every one of them, and stores its parameter values the same way [157]. **Raw video** is an integer array with sidecar metadata: high-speed camera recordings in structural dynamics reach hundreds of gigabytes and are decoded from proprietary formats into a three-dimensional array of images, height and width, read as a memory-mapped unsigned sixteen-bit array beside a dictionary of metadata from a sidecar file [158]. **Point clouds** from 3D scanners are product-quality data: the manufacturing review names coordinate measuring machines, 3D scanners and

computer tomography as the technologies that measure how well parts meet their specification, with one study storing point clouds of individual components and their deviation from nominal [91]. **Images** travel as files: the data-analytics reference architecture publishes camera images as compressed image files on a Kafka topic and consumes them from it [138].

What these share is that the value is not a number but an array, and the tuple's `value` field holds a reference – a file path, an object key – with the array's shape, type and axes in the metadata beside it. The event time, binding and quality fields are unchanged. A thermal image of the bearing housing at 06:10:00 is still a tuple; its value is an object in the store the Object storage subsection describes.

5.11.4 The time-series database

Chapter 2 introduced the shape when it described InfluxDB's point: a **measurement** name, **tags**, **fields** and a **timestamp**. The shape is general to time-series databases, and the reading at 06:10:00 takes it like this:

```
measurement:  bearing_temp
tags:         turbine=wt07   component=gen-bearing   quality=good
fields:       value=61.2
timestamp:    2026-06-09T06:10:00Z
```

The one design decision is which of the tuple's parts became tags and which became fields. Tags are what you filter by, and the store indexes them; fields are what you aggregate, and the store scans them. The binding's parts – turbine, component – are tags, because the questions are "this turbine's bearings" and "all turbines' generator-side bearings". The value is a field. The quality is a tag if you filter on it ("only good readings") and a field if you do not; here it is a tag. The unit is in neither: it is fixed per measurement and lives in the metadata below. A fleet of two hundred turbines with a handful of monitored components each makes the `turbine` and `component` tags a few hundred distinct pairs, and the index stays that size for the life of the fleet.

The high-cardinality tag is the mistake that is hard to reverse. A tag with a new value on every message – a message identifier, the MQTT sample's `seq`, the event time itself – gives the store a new distinct value to index on every write, so the index grows with the data instead of with the fleet: 144 new series per binding per day for this stream, and the store slows and swells until the tag is removed, which means rewriting the data. The test before making anything a tag: count its distinct values, and ask whether the count is the size of the fleet or the size of the history.

5.11.5 Retention and downsampling

A store keeps raw points for a **retention** period and then either drops them or replaces them with summaries – **downsampling**. Chapter 2's rig keeps seven days and drops; a turbine's twin keeps the raw stream for weeks at the edge and summaries for years in the cloud. Turbine owners' second answer to volume, after lossless files, is exactly this: keep simple aggregates such as ten-minute means at the edge, which is fast and compresses well and loses quality with no bound on the loss [156].

The loss that matters is which summary is kept. Downsampling that keeps only the mean destroys the very thing a fault detector needs: a vibration spike that lasted a second is averaged into a ten-minute mean and vanishes, and the record of the month shows a calm turbine. Keep min, max and count beside the mean, so that a spike survives the summary that averaged it away and a gap shows as a count below the expected number of readings. After thirty days the three readings above and the three that followed them become one row for the hour:

```
bearing_temp  turbine=wt07 component=gen-bearing
  mean=61.3  min=61.1  max=61.6  count=6      2026-06-09T06:00:00Z
```

Four numbers per window instead of one, and the month's record can still say that something happened on the ninth: a `max` far above the `mean` is a spike, a `count` of five is a missing reading.

5.11.6 Model-based compression within a bound

There is a third answer to volume, between lossless files and lossy means. **Model-based compression** represents a span of readings by a fitted model – a constant, a line, a small polynomial – within a declared error bound, and stores the model's parameters instead of the points. The three readings climb by a tenth of a degree per ten minutes and a fourth at 61.5 would too; a line through them, stored as a start value, a slope and a span, reproduces all four within the sensor's resolution and replaces four points by three numbers. The time-series system built for turbine data does this at scale: it compresses groups of series within a user-defined pointwise error bound, which may be zero, by trying several model types and keeping the one that compresses best, falling back to a lossless encoding where the values move more than the bound allows [155]. On real turbine data it outperformed the lossless big-data formats even at a zero bound, a tiny bound of one hundredth of a per cent improved compression significantly, and at a five per cent bound it was nearly thirty times smaller than the lossless files [156].

Why a bounded loss can be free: recall Equation 5.1. The ten-bit temperature reading was already rounded to a third of a degree by its converter, so a compression that reconstructs it within a tenth of a degree has lost nothing the sensor ever had. A bounded loss smaller than the sensor's own quantisation step is free, and the datasheet's resolution row is where the bound comes from. The same system's authors note that the edge hardware at renewable installations resembles a low-end commodity desktop computer, so a high compression ratio is worth more there than a low processor load [155] – the Sampling section's ceiling, set by the gateway, showing up in the store.

5.11.7 Object storage: the cold path

The cold path in one paragraph: Parquet files in an object store, partitioned by time and then by asset, with the raw data kept so that it can be reprocessed when a function's version changes. An object store holds named blobs and answers two questions quickly – give me this key, list the keys under this prefix – and nothing else, so the layout of the keys is the whole of its design. The day's readings for the bearing land under

```
telemetry/year=2026/month=06/day=09/turbine=wt07/bearing_temp.parquet
```

with the 144 readings of the day as columns inside it. Partition by time first because every cold-path query is bounded by a span – June, last year – and a time-first layout lets the query skip whole prefixes; by asset second because the next filter is always "this turbine". Keep the raw data because the derive functions will change, and a summary computed by version 3 of the residual cannot be recomputed by version 4 from the summary. Industrial cyber-physical systems have adopted data lakes on distributed file systems, and what those lakes hold is raw data [159]; the turbine systems above write their compressed segments continuously to an object store in the cloud [154]. What an object store cannot answer quickly is "the latest reading", because finding it means listing keys and opening a file; that is the time-series database's question, which is the reason a twin holds both.

5.11.8 Metadata

Every store above holds the readings; none of them holds what the readings mean. The 612 in the register, the 61.2 in the column and the `value` field of the point are the same number, and nothing about any of them says that it is a temperature in degrees Celsius of the generator-side bearing of turbine 7, read every ten minutes by a gateway on network time from a sensor accurate to half a degree and last calibrated in March. That is **metadata**, and for the bearing stream it is one record:

```
{"binding": "wt07/gen-bearing/temp",
 "device": "wt07-plc-1", "register": 42,
 "scale": 0.1, "offset": 0.0, "byte_order": "big-endian",
 "quantity": "bearing temperature", "unit": "degC",
 "range": [-40.0, 150.0], "resolution": 0.1, "accuracy": 0.5,
 "sampling_interval_s": 600, "clock_source": "gateway, network time",
 "calibrated": "2026-03-14", "firmware": "4.2.1",
 "schema_version": 3,
 "valid_from": "2026-03-14T00:00:00Z"}
```

Read it against the chapter: the first three lines are the binding the Transports section's handler applies; the next two are the Sensors section's datasheet numbers, which the validate function reads its range from; the next is the Sampling and Time sections' interval and clock; `schema_version` is what the Encodings section said must be kept when the schema is not in the message; and `firmware` is the Vocabulary section's digital thread, written down. The last line is the one that makes it metadata rather than configuration: a reading is interpreted by the record that was valid at its event time, so when the sensor is recalibrated or the firmware changes, a new record with a new `valid_from` is written and the old one is kept. A reading from March is then still readable in November after both have changed. The record is small, read rarely and versioned in the configuration Chapter 4 put under version control; it lives in a document store or a table keyed by binding, and the arrays of the Arrays and fields subsection carry the same thing as a sidecar beside each object. Without it, every store in this section is a store of numbers.

5.11.9 Sizing the volume

Size the volume before it is a problem, from the sampling interval, the number of bindings and the bytes per point. A published figure gives the arithmetic: a wind turbine generating 2,500 time series sampled at 100 Hz produces over 321 GiB per day if timestamps and values take eight bytes each [156]. Check it:

$$V_{\text{day}} = 2500 \times 100 \times 86{,}400 \times 16 = 3.456 \times 10^{11} \tag{5.4}$$

bytes, which is 321.9 GiB. Equation 5.4 is one turbine, one day, raw. Now set it against the link. The same paper reports that bandwidth between edge and cloud can be as low as 512 kbit/s [156]; a day's raw data is $3.456 \times 10^{11} \times 8/86{,}400 = 3.2 \times 10^{7}$ bits per second averaged over the day, thirty-two megabits a second against a link of half a megabit – more than sixty times too much. That single division decides the architecture: the raw high-rate streams cannot leave the turbine, so the edge stores them, compresses them within a bound, and sends segments; the twin in the cloud sees compressed segments and summaries, and the faded turbine example works the consequence. Do this division for every twin on the day the sampling intervals are chosen, because it is the cheapest architectural decision in the chapter and the most expensive to discover late.

5.12 Quality: The Field the Pipeline Carries

The tuple's sixth field has waited until the pipeline that fills it was built. **Quality** is the adapter's statement about how far the value may be trusted, and it takes one of six states.

5.12.1 Six states

Table 5.8 names the six and what each does downstream, in the three components that read it: the estimator of the state and estimator chapter, the calibration that fits the model to the data, and the alerting service.

Table 5.8: The six quality states and what each does in the three components that read it. The fourth and fifth rows say nothing about the asset and everything about the instrument; the last two are different failures pointing at different teams.

State	Meaning	Estimator	Calibration	Alerting
`good`	measured, in range, on time	uses it	fits to it	alerts on it
`uncertain`	measured, but the sensor is suspect: drift, disagreement, stuck	uses it with a wider variance	excludes it	alerts, with the state shown
`substituted`	filled in, not measured: an interpolation, a last-known value	uses it as a prior, not an observation	must exclude it	never alerts on it alone
`out of range`	measured, outside the sensor's stated range	ignores the value, notes the event	excludes it	alerts on the sensor, not the asset
`no reading`	the adapter asked and nothing came back	none; the age of twin grows	none	alerts on the link or sensor
`not attempted`	the adapter did not ask: not bound, paused, misconfigured	none	none	alerts on the twin itself

Two rows need a sentence. A substituted value must never lose its label. The moment a filled-in number is stored as if it were measured, the calibration that later fits to it is fitting to the twin's own guess, and the model learns to predict its own interpolation. The clean function may fill a gap; it may never fill it with a `good`. And `no reading` and `not attempted` are different failures pointing at different teams: the first says the sensor or the link failed and belongs to whoever owns the hardware; the second says the adapter never asked – the binding is unbound, the poll loop is paused, the configuration names a sensor that was removed – and belongs to whoever owns the twin. A monitoring service that merges the two into "missing" has thrown away the one fact that says who to call.

5.12.2 What the adapter can decide

The adapter can decide quality from two sources and no more: the sensor and the transport. From the sensor it can read a status register or a confidence the sensor itself reports; from the transport it can see a timeout, a checksum failure, a sequence gap, a will message. Those give it `out of range`, `no reading`, `not attempted`, a `substituted` label on anything the clean function filled, and `uncertain` where a second sensor disagrees or a calibration date has passed. What it cannot decide is whether a valid, in-range, on-time reading is *plausible*, because plausibility needs a model of what the reading should have been – the stuck sensor of the Functions section, the temperature that rose faster than any bearing can. That quality is decided in the core, by the state and estimator chapter, and the adapter's honest state for a reading it cannot judge is `good`, meaning "nothing I can see is wrong".

5.12.3 Gaps, duplicates, out of order

Three defects in the stream itself are the adapter's and the clean function's to detect, and Table 5.9 gives each its detection and its cost.

Table 5.9: Gaps, duplicates and out-of-order arrival: how each is detected in the adapter or the clean function, and what each costs when it is not. All three are properties of the transport that the tuple's event time and a sequence number make visible.

Defect	How it is detected	What it costs if missed
Gap	a sequence number skips; the interval between event times exceeds T_s by more than the slack; a heartbeat is missed	a window computed over fewer readings than it claims; a trend fitted across a hole
Duplicate	two tuples with the same binding and event time; a repeated sequence number after a reconnect	every count and sum doubled for the span; a spike that is only a retry
Out of order	an event time earlier than the last one stored for the binding	a correction applied before the reading it corrects; a rate of change with the wrong sign

The operational-metrics study of the Time section is the experimental form of this table: packet loss with jitter, replay and out-of-order delivery each left a measurable mark on the twin's timeliness, reliability and consistency metrics [113]. The sensor-side counterpart is the drift and calibration trouble of the Sensors section, which reaches the store as noise and inconsistency [140] and is the reason `uncertain` exists.

5.13 How to Choose

Everything above collapses into one table and one checklist. The table is organised by the path of the Vocabulary section, because the path is the decision that fixes the rest.

Table 5.10: How to choose, by path. Assign a service to a path and read down its column; every row is a decision the path has already made.

	Hot path	Warm path	Cold path
Latency budget	milliseconds to seconds	seconds to minutes	hours to months
Sensing rate	at the ceiling the event allows	derived from the check interval	whatever the hot and warm paths produced
Transport	a broker at the edge, or a direct subscription	a durable log	files: Parquet segments over a metered link
Encoding	binary, schema-required	binary or JSON, with a sequence number	columnar files
Layout at rest	rows: the latest reading per binding	rows and short windows	columns, partitioned by time then asset
Placement	edge	edge or fog	cloud
Store	in memory, or a time-series store with short retention	a time-series store with downsampling	an object store, raw kept
Late data	drop	grace period	emit corrections

Read Table 5.10 down a column, not across a row: a hot-path alarm is a binary message on an edge broker into an in-memory latest-value store with late data dropped, and none of those choices needs separate argument once the path is chosen.

Then the checklist, in order of leverage – each question changes more of the design than the one after it:

1. **What is the latency budget?** The decision's latency from Chapter 1, which chooses the path, which chooses the column.
2. **Who initiates?** Whether the source can push, or must be polled, and whether either is resumable.
3. **How many consumers, and do they grow?** One consumer can poll; ten need a broker; ten that will become a hundred need a durable log.
4. **What are the power and bandwidth?** A battery-fed gateway and a metered link push toward binary encodings, long sampling intervals and compression at the edge.
5. **What must the payload mean to other systems?** If another team's software will read the messages, the encoding must be self-describing or the schema must be published, and the units must be a code list.

Two cautions close the section. Published protocol benchmarks use workloads unlike real devices: the benchmark literature's own critique is that existing publish/subscribe benchmarks stress systems with static configurations of homogeneous publishers, where real IoT devices publish at heterogeneous rates, with varied payload sizes, under intermittent connectivity [160]. And latency is a property of the deployment, not of the protocol: a measurement study across the network stack found that core routing, the access network, the transport protocol and the application protocol each contribute, that a new connection costs a handshake of about eighteen

milliseconds and a security setup of about twenty-eight more, and that moving computation close to the user is only a small part of the latency problem [161]. The same protocol on the same hardware gives different numbers on a different network. Measure your own.

5.14 The Plant Demonstrator's Data Path

Everything above used the turbine, a building or a generic instance. This section applies the chapter, in order, to the one system the previous chapters put on the desk, and it is the only place the plant demonstrator appears in this chapter. It is built against Chapter 2's API and Chapter 3's stages, and it makes no claim about the rig beyond what Chapter 2 established.

5.14.1 The rig's sensors, read as a designer

Recall Chapter 2's table of quantities. The STEMMA soil sensor reports a ten-bit count scaled linearly to a percentage, every $T_{\mathrm{r}} = 10$ seconds in the shipped configuration; the light sensor reports nine wavelength bands every fifteen seconds; the air sensor reports temperature and humidity at a configurable interval, with a stated confidence of plus or minus 0.2 degrees; the RS-485 probe reports moisture, temperature and conductivity from three registers; and the controller records a `watering` event per dose. Read those as a designer with the Sensors section in hand.

The soil sensor's per cent is a relative scale, as Chapter 2 warned: its range is the converter's ten bits, its resolution is one count in 1,024, and its accuracy against water content is unstated, so the datasheet's accuracy row is empty and the `uncertain` state is the honest quality for it until the calibration chapter fills that row. Its binding is not one address but a pair – the multiplexer's channel, then the sensor's address – so the multiplexer channel is part of the binding, and a sensor moved to another channel is a new binding, not a changed value. One poll of the light sensor produces nine tuples, one per band, sharing one event time. And the watering events are ingest, not actuation: the controller records what it did, and the twin reads that record through the same API as a reading, with quality `good` and a unit of millilitres. Chapter 2's warning stands – that column is a record of the pump schedule, not of the pot.

5.14.2 The REST API against the five properties

Score the API of Chapter 2 against Table 5.4. Event time preserved: yes, every point carries the `time` the controller stored it at, which is the gateway record of the Time section and the best event time the rig offers. Connection state observable: yes, every `GET` returns a status code or fails, and `GET /` is a liveness check. Gap survivable: yes, `since_timestamp` on the sensing and watering endpoints returns everything since a given time. Ordering knowable: yes, points come newest first with their times. Deduplication possible: yes, unit, parameter and time identify a point. Five out of five, and the one parameter that earns the score is `since_timestamp`: it makes polling resumable, so a twin that was down for an hour asks for the hour and gets it – within Chapter 2's seven-day retention, which is the one limit on the property.

5.14.3 Bindings and the tuple mapping

Table 5.11 maps the rig's quantities to bindings and tuples.

Table 5.11: The rig's bindings and the tuple each produces. The multiplexer channel is inside the first binding's configuration, not in the tuple; the nine light bands are nine bindings sharing one poll.

Binding, and the endpoint behind it	Quantity and unit	Event time	Quality the adapter can decide
soil moisture of `plant_1`, from `/sensing/plant_1/moisture`	soil moisture, relative, in per cent of full scale	the point's `time`	`uncertain` (uncalibrated scale)
soil conductivity of `plant_1`, from the `soil_electrical_conductivity` parameter	conductivity, in microsiemens per centimetre	the point's `time`	`good`
air temperature of the greenhouse, from the `air_temperature` parameter	air temperature, in degrees Celsius	the point's `time`	`good`, with the stated confidence carried
nine light bands of the greenhouse, from `light_level_415nm` and eight more	photon flux per band, in photons per second	one `time` for nine tuples	`good`
watering of `plant_1`, from `/actuation/plant_1/watering_events`	dose delivered, in millilitres	the event's `time`	`good`; an event, not a sample

5.14.4 The encoding at each hop

The soil reading is a ten-bit integer in an I2C register – a number at an address, schema in the driver. On the Raspberry Pi it becomes a Python float, then an InfluxDB point with the value, its units and a confidence as fields. Through the API it becomes a JSON object, `{"time": ..., "physical_unit": "plant_1", "parameter": "moisture", "value": 41.2, "units": "%", ...}` – self-describing, with the unit as a field, which is why the adapter can validate it. In the adapter it becomes the tuple. Three encodings, one scale factor, applied once, on the Pi; the adapter applies none, and its transform function for this binding is the identity.

5.14.5 The poll loop and its watermark

The adapter polls each binding at its own interval with `since_timestamp` set to a **watermark**: the event time up to which everything has been durably stored in the twin's own store. The loop is: read since the watermark; validate each point; write the accepted tuples to the store; and only when the write has succeeded, advance the watermark to the newest event time written. A watermark that advances on a successful read loses every reading the write then fails to keep. The loop is resumable by construction – after an outage it asks since the old watermark – and idempotent, because a point returned twice has the same unit, parameter and time and the clean function drops the second.

5.14.6 The numbers

Table 5.12 sizes the path for one pot, from Chapter 2's intervals and an assumed sixty-second air interval, with a JSON object of about 150 bytes per point.

Table 5.12: The rig's data path for one pot and one day. Ten megabytes a day is a hundred kilobits a second averaged over the day – nothing, on a desk network – and the light sensor is four fifths of it.

Quantity	Interval	Points per day	Bytes per day, as JSON
soil moisture	10 s	86,400/10 = 8640	about 1.3 MB
light, nine bands	15 s	$5760 \times 9 = 51{,}840$	about 7.8 MB
air temperature and humidity	60 s, assumed	$1440 \times 2 = 2880$	about 0.4 MB
watering events	per dose	two or three	negligible
total for one pot		about 63,000	under 10 MB

Two more numbers. The age of twin, by Equation 5.3, depends on which reading the state rests on: if the adapter polls the moisture every $T_{\mathrm{r}} = 10$ seconds, A sits between zero and about ten seconds plus the poll's latency, and a fault detector checking every $T_{\mathrm{c}} = 6$ hours has a state that is at most a few seconds old when it reads it. That is far better than the decision needs; the sampling section's five steps would say the moisture could be read every few minutes with no loss to the decision, and the aliasing check has not been run to say so.

5.14.7 Limitations the credibility chapter will inherit

Five, stated plainly. The soil sensor's accuracy against water content is unknown, so every moisture tuple is `uncertain` until calibrated. No aliasing check has been run on any of the rig's quantities; the ten- and fifteen-second intervals were chosen as reasonable, not derived. The event time is the controller's record time, not the sensor's read time, and the gap between them is unmeasured. A stuck soil sensor is undetectable on this path, because nothing here has a model of how fast a pot can dry. And the seven-day retention below the API bounds the gap the twin can survive. Each of these is a sentence the credibility chapter will have to carry into its argument about how far the rig's twin can be believed.

5.15 Faded Examples

Three, decreasing in how much is worked, none of them the demonstrator.

5.15.1 Example 1, worked in full: the offshore turbine's ingest

The turbine that has carried this chapter, made concrete. It is consistent with the models chapter's running example, which uses the same turbine.

Three sampling rates and a fourth timescale. The turbine's SCADA system reports operational signals – wind speed, power, rotor and generator speed, and the temperatures of the shaft bearing, brakes, generator and gearbox oil – at a rate that averages a few tenths of a hertz on a research turbine and is often available only as ten-minute statistics on a commercial one [22, 162]. Its condition-monitoring system samples bearing and gearbox vibration at hertz to tens of hertz. Its structural sensors – tower and blade accelerometers – sample at kilohertz. And its fatigue summary is computed once a day. Four timescales, four decisions: a ten-minute mean serves "is the turbine producing as expected", a hertz stream serves "is a bearing degrading over weeks", a

kilohertz stream serves "did the tower exceed its design load in this gust", and the daily summary serves "when does this turbine need inspection".

Clocks. Apply the ratio of the Time section. The ten-minute SCADA statistics tolerate a clock a second wrong. The two-kilohertz accelerometers do not: one second is two thousand samples, and comparing two accelerometers to find a vibration's direction needs the two clocks within a sample of each other, which is hundreds of microseconds. Network time cannot hold that; precision time over the turbine's wired network, or a satellite reference at the gateway, can. That is a purchase, and this paragraph is where it is justified.

Volume and the boundary. By Equation 5.4, a turbine with a few thousand series at a hundred hertz produces over 321 GiB a day, against an edge-to-cloud link that may be as slow as half a megabit a second [156]. The high-rate streams do not leave the turbine intact. So the boundary sits at the edge: the gateway in the turbine ingests everything, runs the hot-path alarms locally, compresses the streams within a bound set from each sensor's resolution, and sends compressed segments and per-window summaries to the cloud as Parquet, where the turbine owners' object store already lives [154]. The twin in the cloud sees the ten-minute statistics in full, the hertz streams within their bound, and the kilohertz streams as summaries and as segments it can request. The raw kilohertz data stays on the turbine for as long as its disk allows, and a request for it is a cold-path job.

Scoring the transports. The SCADA export is a CSV file per day: gap survivable and ordered, connection state unobservable, deduplication by timestamp. The condition-monitoring system publishes MQTT with a sequence number and a timestamp in every payload and a will message registered at connection: all five, by convention. The accelerometers write to the edge's local store directly: all five, because the adapter is the writer. Three transports, one tuple, and the age of twin is computed for each.

5.15.2 Example 2, one step left to you: a battery bus fleet

A city runs two hundred electric buses. Each bus's battery management system takes voltage, current and temperature from every cell in the pack every one to five seconds and estimates the pack's state of charge and health, neither of which can be measured directly [163]. The buses are mobile and intermittently connected: in the depot they have wireless network access, on the route they have a metered cellular link that drops in tunnels and at the edge of the city.

The data path is mostly settled by those two facts. The bus is the edge, and it stores everything it measures; the transport is store-and-forward, with the bus publishing to a depot broker whatever it could not send on the route; late data is therefore central rather than exceptional, and every window in the fleet twin must have a grace period or emit corrections; and the cell voltages are the smooth, high-volume channel – a few hundred cells at a hertz – for which model-based compression within a bound set by the voltage sensor's resolution is the right answer, while the current, which spikes with every acceleration, is kept at full rate.

Your step: the fleet twin's state-of-health estimate is recomputed nightly from the day's data, and a bus that was out of coverage for the last hour of its route uploads that hour at 01:00, after the nightly computation has run. Say which late-data policy the nightly computation must use, what the store must keep for that policy to be possible, and what the age of twin for that bus should have shown at midnight. A check appears in the exercise solutions.

5.15.3 Example 3, posed as a problem: a district heating network

A district heating utility has three generations of equipment from three vendors. The newest substations publish MQTT with timestamps from gateways whose clocks are known to drift by minutes. The older substations speak MODBUS to a SCADA system that will export a CSV file

of ten-minute averages once a day. The oldest have no telemetry at all, and a technician reads their meters by hand once a quarter and types the readings into a spreadsheet. The utility wants one twin of the network.

Design its upward leg, in eight parts: (1) the tuple that all three sources are normalised to, and what the quality field says for each; (2) the event time each source can offer, and which of the four timestamp questions each fails; (3) how the MQTT clock drift is detected and what is done with a reading whose event time is impossible; (4) the transport score of each source against the five properties, and what convention or component repairs each missing one; (5) the sampling interval each source imposes, and which decisions of the utility's each interval can and cannot support; (6) the store and layout for the quarterly manual readings, and whether they are sampled data or events; (7) the store's schema – which parts of the binding are tags, and what retention each source needs; (8) the age of twin per substation, and what the twin's dashboard should show for a substation whose newest reading is three months old.

5.16 Summary

Back to the objectives this chapter opened with.

1. **Read the datasheet for six numbers.** Range, resolution, accuracy, drift, response time and maximum sample rate, each changing something downstream; resolution is how finely a sensor reports and accuracy is how far it may be wrong, and a threshold finer than the accuracy fires on the sensor. Calibration is the number the datasheet cannot give and the cost most often skipped.
2. **Derive the interval from the decision.** Five steps – the decision, the event, several samples across it, the ceiling, the floor – give the sampling interval; aliasing is fast behaviour appearing as slow behaviour, checked by sampling fast once and decimating; the theorem fits on one line; quantisation puts a floor of half a count under every use of a reading. Sample the state, capture the events, align them on time.
3. **Ask five things of a transport.** Event time preserved, connection state observable, gap survivable, ordering knowable, deduplication possible – and before all five, can I ask you for what I missed. HTTP with a history parameter, MQTT with a sequence number and a will message by convention, OPC UA with its address space and two timestamps, and Kafka with its offsets each answer differently; MODBUS carries two bytes and needs a gateway to answer for it. One handler per protocol, one tuple upstream, and the tuple of six fields is the adapter's output contract.
4. **Carry two times and compute three numbers.** The tuple carries event time, the adapter records ingest time, and their difference is a measurement; clock error matters as a fraction of the sampling interval; twinning rate, age of twin and time discrepancy per binding say whether the twin is current, how current, and whose fault it is not. Age of twin is the best single health number a twin has, and `as_of` is what makes it computable.
5. **Encode in flight, count the bytes.** A register is three conventions applied once; one reading is 124 bytes as JSON, 101 as CBOR, 42 as Protocol Buffers, 37 as Avro and 2 as a register, and every byte saved is a fact moved into a schema somebody must keep; encodings are self-describing or schema-required, and the choice is about which is scarce, bytes or engineering time; units travel as a code or cannot be checked.
6. **Sort the functions, then build five of them.** Shadowing computes state from readings and a model, augmentation computes what the asset cannot about itself, memorisation keeps the history; this chapter's ingestion functions are the part of shadowing that runs before any model. Ingest, validate, clean, transform, derive fail by silence, refusal, invention, double application and version drift; quarantine rather than discard; the adapter runs

what needs only the binding and the core runs what needs the state; test with a reading and a number, version with the configuration. The stuck sensor is the check no adapter function can make.

7. **Lay the readings out five ways and size the volume first.** Rows for the latest, columns for the history; tags are filtered, fields are aggregated, and a high-cardinality tag is hard to undo; keep min, max and count beside the mean; a bounded loss smaller than the sensor's quantisation is free; Parquet in an object store keyed by time then asset, raw kept; a versioned metadata record beside it all, without which every store is a store of numbers; and one division – bytes per day against bits per second on the link – decides where the boundary sits. On the rig, the whole path is ten megabytes a day and `since_timestamp` earns the API five out of five.

The sentence to carry forward: **every number the twin will later reason about is a tuple this chapter's components produced, and a large part of whether the twin deserves to be believed is settled before any model runs.**

5.17 Exercises

Solutions or hints follow the exercises; each says which. The objective each exercise serves is given in brackets. None of the exercises uses the demonstrator.

Exercise 1 (Objective 1, easy). A datasheet for a pressure sensor gives: range 0 to 10 bar, resolution 0.01 bar, accuracy plus or minus 0.1 bar, drift 0.05 bar per year, response time 50 ms, maximum sample rate 100 Hz. A colleague proposes an alarm when the pressure changes by more than 0.05 bar between consecutive readings. Say what is wrong with the proposal, and what the smallest defensible threshold is.

Exercise 2 (Objective 2, easy). A cold-storage room's compressor cycles every 20 minutes. Its temperature is sampled every 20 minutes. Say what the record will show, then propose two different sampling intervals that would show the true cycle, and say which of the two you would use and why.

Exercise 3 (Objective 2, medium). A twelve-bit converter reads a sensor that outputs 0 to 5 V over a level of 0 to 2 metres of water. Compute the quantisation step in metres and the worst-case rounding error. Then say whether a compression bound of 1 mm would lose anything the sensor had.

Exercise 4 (Objective 4, medium). A gateway's clock is 3 seconds fast. For each of three bindings – a room temperature every 5 minutes, a motor current every 100 ms, and a power-quality waveform at 10 kHz – compute the clock error as a fraction of the sampling interval and say whether network time synchronisation is enough.

Exercise 5 (Objective 3, medium). Score each transport against the five properties, giving a yes, a no, or "by convention, if the payload carries X": (a) a vendor's REST endpoint that returns only the latest value; (b) bare MQTT with quality-of-service level 1; (c) MQTT whose publisher puts a sequence number and an event time in every payload and registers a will message; (d) a nightly CSV file named by date; (e) a Kafka topic written by the gateway.

Exercise 6 (Objective 4, medium). A twin's state document shows `as_of` 06:10:00 and the wall clock reads 06:47:30. The binding's sampling interval is 10 minutes and the configuration's staleness bound for issuing a command is 15 minutes. Compute the age of twin, say what the lifecycle field should read, and say what a command-path check does.

Exercise 7 (Objectives 5 and 7, medium). For each data shape, choose an encoding in flight and a store and layout at rest, and say what each preserves and loses: (a) forty temperatures per second from a kiln over a metered satellite link; (b) a maintenance report typed by a

technician; (c) a thermal camera's frames once a minute; (d) the record that says which sensor sits on which pump of a pumping station, with its register, scale, unit and calibration date.

Exercise 8 (Objective 6, medium). Write, in the shape of the validate listing, a clean function for a binding whose readings arrive over MQTT with a sequence number in the payload. It must detect a gap, a duplicate and an out-of-order tuple, and say what quality state, if any, it attaches in each case. Then write the three tests, each a tuple and an expected outcome.

Exercise 9 (Objective 7, medium). A fleet of 300 pumps each has 12 sensors sampled every second, with 16 bytes per point. Compute the raw volume per day for the fleet. Then propose a retention and downsampling policy with three tiers, say which summaries the second tier keeps and why, and compute the volume of the third tier after a year.

Exercise 10 (Objective 7, medium). A colleague proposes a time-series schema with tags for site, pump, sensor, firmware version, and the sequence number of each message. Say which tag is the mistake, why it is hard to reverse, and where that value belongs instead.

Exercise 11 (Objectives 3 and 4, hard). Finish Example 2 of the Faded Examples section: the late-data policy for the nightly state-of-health computation, what the store must keep for it, and the age of twin at midnight for the bus that was out of coverage.

Exercise 12 (Objectives 1 to 7, hard, open-ended). Do Example 3 of the Faded Examples section, the district heating network, in all eight parts.

5.17.1 Solutions and hints

Exercise 1 (solution). The threshold is below the sensor's accuracy (0.1 bar) and at half its drift per year; a change of 0.05 bar between readings is inside the sensor's own uncertainty and the alarm fires on the instrument. The smallest defensible threshold is above the accuracy – a change of more than 0.1 bar, and in practice a multiple of it – and it should be a change sustained over several readings, since response time and noise make a single-step test unreliable.

Exercise 2 (solution). The record shows a nearly constant temperature, or a slow phantom drift as the sample instant walks round the compressor's cycle – aliasing. Any interval well under half the cycle shows the true cycle: every 2 minutes gives ten samples per cycle; every 5 minutes gives four. Use 2 minutes if the decision is about the compressor's health (the shape of the cycle matters) and 5 minutes if the decision is about the room's mean temperature (an average over the cycle is enough); either way run the check – sample every 10 seconds for a day and decimate – before trusting the choice.

Exercise 3 (solution). $2^{12} = 4096$ counts across 5 V is 1.22 mV per count; across 2 metres it is $2/4096 = 0.00049$ metres, about half a millimetre per count, and the worst-case rounding error is a quarter of a millimetre. A compression bound of 1 mm is two counts, larger than the quantisation step, so it loses something the sensor had – a small something, and defensible, but not free. A bound of 0.2 mm would be free.

Exercise 4 (solution). Room temperature: $3/300 = 0.01$, one per cent of the interval; network time is ample. Motor current: $3/0.1 = 30$ samples displaced; network time, which holds tens of milliseconds, is enough for the interval but the 3-second error itself is not acceptable and says the gateway is not synchronised at all. Waveform: $3/0.0001 = 30{,}000$ samples displaced; network time cannot serve this binding and a precision time source at the gateway is required.

Exercise 5 (hint). (a) None of the five for free; event time only if the payload carries it; not resumable, so a gap is unrecoverable. (b) Delivery at least once gives duplicates with no way to tell them apart unless the payload carries a sequence number; connection state only via the will; no gap recovery. (c) Four by convention – the will message for state, the event time for time, the sequence number for ordering, duplicates and the visibility of a gap – and gap recovery only if the publisher also keeps a buffer it can be asked to resend from. (d) Gap survivable and ordered by construction; connection state unobservable until the file fails to appear; event time preserved

if the rows carry it. (e) Ordering, deduplication and gap recovery from the log's offsets; event time and connection state still depend on what the gateway wrote.

Exercise 6 (solution). The age of twin is 37 minutes 30 seconds. Against a 10-minute interval that is nearly four missed updates, so the lifecycle field should read *out of sync*; against a 15-minute staleness bound the command path must refuse, hold the last safe command, and raise the inward alert – Chapter 3's staleness row, computed from Chapter 4's `as_of` by this chapter's Equation 5.3.

Exercise 7 (hint). (a) A binary schema-required encoding with a sequence number, compressed at the edge within the sensor's resolution; columnar at rest; loses nothing the sensor had, costs a versioned schema. (b) Self-describing text, since a person wrote it and a person will read it; a document store; loses nothing, gains nothing from compression. (c) The frame as a file with a JSON sidecar; an object store; the tuple's value is the object key. (d) Metadata: a self-describing JSON record per binding with a `valid_from`, in a document store or a table keyed by binding, versioned rather than overwritten; loses nothing, is read rarely, and is what makes every other store's numbers mean something.

Exercise 8 (hint). Keep the last sequence number and last event time per binding. A sequence number more than one ahead is a gap: pass the tuple, record the gap, and mark the window's count short. The same sequence number twice is a duplicate: drop the second, attach nothing. An event time earlier than the last stored is out of order: buffer it for the grace period and insert it in order, or emit a correction if the window has closed. Only a filled value gets `substituted`; the other two attach no quality state, because the tuple itself is good – it is the stream that had the defect.

Exercise 9 (solution). $300 \times 12 \times 86{,}400 \times 16 = 4.98 \times 10^9$ bytes a day, about 4.6 GiB for the fleet. Three tiers: raw for 30 days; one-minute windows with mean, min, max and count for two years; hourly windows with the same four for the asset's life. The second tier keeps min and max so a spike survives and count so a gap is visible. The third tier after a year: $300 \times 12 \times 8760$ windows of four 8-byte values plus a timestamp, about $3.15 \times 10^7 \times 40 = 1.26 \times 10^9$ bytes, a little over a gigabyte – four thousand times smaller than the raw year.

Exercise 10 (solution). The sequence number: its distinct values grow with every message, so the index grows with the history rather than with the fleet, and removing it later means rewriting every point. It belongs as a field, or better in the clean function's own state, since its only job is gap and duplicate detection. Firmware version is a legitimate tag – a handful of values per fleet – and the digital thread wants it there.

Exercise 11 (solution). The nightly computation must emit corrections: it runs at midnight over the day's data, and when the missing hour arrives at 01:00 it recomputes that day's state of health and publishes it again, marked as superseding the first. For that to be possible the store must keep the raw cell data for at least the day, not only the nightly result, and the result must carry the version and the data span it was computed from. At midnight that bus's age of twin should have read about one hour and its lifecycle *out of sync*, and a dashboard showing its state of health with that age beside it is telling the truth.

Exercise 12 (open). Assessed on whether the three sources reach one tuple with three different quality vocabularies rather than one; on whether the quarterly readings are treated as events with a stated uncertainty rather than as samples; on whether the MQTT clock drift is detected from the time discrepancy and impossible timestamps are quarantined rather than corrected; and on whether the dashboard for the oldest substations shows an age of twin of three months rather than a number.

5.18 Further Reading

- *Sensing and sampling:* [139] for a published twin whose sensor array is specified with ranges and accuracies, the real form of the datasheet table; [140] for calibration and wear as sources of bad data reaching a twin; [91] for the distinction between a sensor's sampling frequency and the twinning rate. The sampling theorem, aliasing and quantisation are textbook material and the mathematics appendix has the argument.
- *Time and synchronisation:* [141] for time discrepancy between digital and physical twins studied in its own right; [113] for timeliness, reliability and consistency metrics as indicators that something is wrong; [121] for twinning rate and age of twin defined as platform metrics.
- *Transports and industrial protocols:* [80] and [142] for HTTP and MQTT side by side and for MQTT's broker features; [143] and [25] for MODBUS and OPC UA as twins actually use them; [144], [145] and [91] for OPC UA's information model, address space and two messaging patterns; [146] for Kafka against other publish/subscribe systems; [83] for one handler per protocol at scale; [137] for heterogeneity as middleware's problem. The protocol samples are this book's own construction.
- *Encodings:* [121] for the encoding family and payload size as latency; [147] for OPC UA PubSub with a binary and a JSON encoding over UDP and MQTT; [148] for the UNECE unit codes as a code list. The six encodings of one reading and their byte counts are this book's own construction.
- *Functions:* [149] for the properties of a twin – reflection, entanglement, memorisation, composability, augmentation and predictability – that the platform literature builds on; [150] and [115] for the shadowing function computing state and propagating actions; [83] for augmentation and composition as the capabilities a passive twin forgoes; [19] for edge functions in the Industrial Internet Reference Architecture; [151] for function as a service.
- *Stream processing and placement:* [138] for hot, warm and cold paths and Kafka as the ingest stream; [150] and [112] for functions relocated along the continuum; [152] for component placement measured on a testbed; [153] for when offloading to the cloud wins.
- *Storage:* [155], [156] and [154] for model-based compression within an error bound, the turbine volume and bandwidth figures, Parquet and Arrow in an edge-to-cloud pipeline; [159] for data lakes that keep raw data; [106], [157], [158], [91] and [138] for NetCDF, labelled arrays, video arrays with sidecar metadata, point clouds and images on a topic. The metadata record is this book's own construction.
- *Data quality:* [113] for the experimental form of the gaps, duplicates and out-of-order table; [140] for the sensor-side causes. The six quality states are this book's own construction.
- *The lifecycle view:* [129] for the digital thread as the Digital Twin Consortium defines it and [17] for the twin as shadow of the present and thread of the past.
- *The faded examples' sources:* [22] and [162] for what a turbine's SCADA system reports and at what rate; [163] for what a battery management system measures per cell and how often.

Chapter 6

Models

6.1 Before You Start

What you are assumed to know. You write software for a living, you read a function signature and know what its arguments and return value commit it to, and you remember basic probability from a first course: what a probability is, what "given" means, and that a distribution is a spread of values rather than one value. You have read the previous five chapters. From Chapter 3 you have three words, recalled here in one clause each: a **model** is a description of how something behaves, and it does not run; a **simulation** is one execution of a model over time from a starting state; a **simulator** is the software that performs simulations. From Chapter 4 you have the seven building blocks, in particular the **models** block, which that chapter gave a slot behind an interface – given a state and a horizon, return a predicted state – and the **state** block, the twin's current belief about the physical twin. From Chapter 5 you have the tuple, the sampling interval T_s and the event time, and the offshore turbine whose generator-side bearing temperature ran through that chapter as its worked stream. From Chapter 2 you have the plant rig's four equations and its dose rule, which reappear once, in the one section of this chapter where the rig is allowed.

What you are not assumed to know. Calculus, linear algebra, numerical methods, any modelling tool and any control theory. Every result this chapter leans on from those subjects is recalled in one clause and pointed at the section of the mathematics appendix that builds it from a loop, a record or a table: *Quantities, Rates and Equations* for what a rate is and what an equation with a rate in it says; *From an Equation to a Loop* for Euler's step and what a solver does; *Many Quantities at Once* for vectors, matrices, the state-space form and modes; *Probability and Distributions* for Bayes' rule and the Gaussian; *State Machines with Probabilities* for the Markov chain and the hidden state seen through a noisy sensor; and *What a Solver and a Simulator Do* for the event queue and the real-time factor. If a paragraph below feels like it is skipping a step, the appendix section named beside it is where the step is.

What this chapter covers. Chapter 4 named the model as one of seven blocks and gave it a slot: given a state and a horizon, return a predicted state. This chapter is about what goes in that slot. It first says what every model has – a state, an input, a parameter set, a step function and an output function – and gives the four criteria a model inside a twin is judged on. It then introduces one running example, an offshore wind turbine, and models it five ways, once per **model type**: a lumped first-principles model of the rotor and the gearbox oil; a state machine of its operating modes and a discrete-event model of the vessel that repairs it; a finite-element model of its tower and blades; a reduced and a surrogate version of that model; and a probabilistic model of a wearing bearing, updated by Bayes' rule. Each model type closes on the same two subheadings, what it costs and how it is kept current, and one section then generalises the second of those into the three rates at which a twin's model changes. A choosing table, the rig, five faded

examples, a summary, exercises and further reading close the chapter.

What this chapter deliberately does not cover. How a solver chooses its step, which belongs to the simulators half of this part and to the appendix, and co-simulation, which belongs with the interface standards it is built on to the standards chapter. State estimation – how the twin corrects the model's state from readings – which is the state and estimator chapter's; this chapter hands it the discrete state-space form and stops. Neural networks, hybrid physics-plus-network models, generative models and foundation models, which are parked for the AI chapter and named here only as parked. Whether a model deserves to be believed, which is the credibility chapter's. Structural change to the model set, which belongs to configuration and the manager chapter.

Learning objectives. By the end of this chapter you will be able to:

1. **Name** the state, inputs, outputs and parameters of a described system and **write** its step function.
2. **Classify** a modelling need onto the five model types and **defend** the choice against the four criteria.
3. **Turn** a rate rule into a loop and **say** what the step size costs.
4. **Explain** in words what a mesh, a mode and a reduced basis are and **judge** when each is worth paying for.
5. **Write** a degradation model as a random process and **update** one of its parameters from a reading with Bayes' rule.
6. **State** the three rates at which a twin's model changes and **name** which chapter owns each.

Every section that follows serves at least one of those six, and the exercises at the end say which.

6.2 What Every Model Has

Every model in this chapter, whatever its physics and whatever its size, is the same five things to a software engineer. Figure 6.1 draws them, and this section names them, because once the five are named the five model types that follow are five ways of writing one of them.

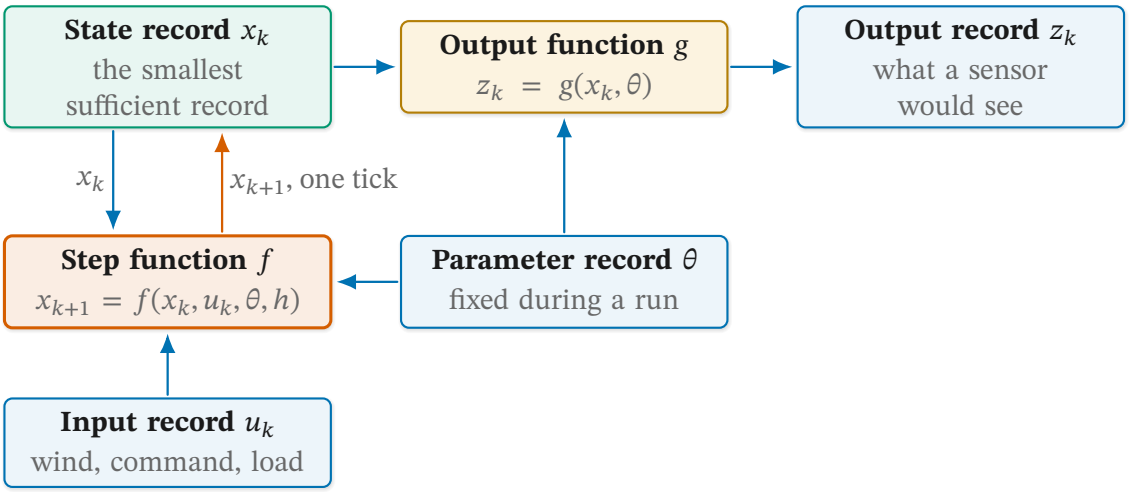


Figure 6.1: The anatomy of a model. Four records and two functions: the step function advances the state record by one interval from the input and the parameters, and the output function reads from the state what a sensor would report.

6.2.1 Four records and two functions

The state record x is the smallest set of numbers that, together with everything done to the system from now on, fixes everything the system will do from now on. That definition is worth reading twice, because it is a test, not a description. A turbine's rotor speed is state: if you know it now and you know the wind and the generator command from now on, you know the speed from now on. Last week's mean wind speed is not state: knowing it adds nothing once the present speed is known. The word is the same as Chapter 4's **state** block, and the relation is exact: the model's state is what the twin's state block holds an estimate of.

The input record u is what is done to the system from outside during a run – the wind, a command from the grid operator, a load on a bridge, a dose of water. The model reads inputs; it never predicts them. A quantity the model predicts is state; a quantity the model is told is input. The same physical thing can be either, depending on where the model's boundary is drawn: the wind is an input to a turbine model and a state of a weather model.

The parameter record θ holds the numbers that stay fixed during one run and may change between runs – an inertia, a resistance, a heat-transfer coefficient, a wear rate. The distinction from state is one of time scale and of who changes them: a solver changes the state every step; an engineer, or later in this chapter a Bayesian update, changes a parameter between runs. Chapter 2's drying rate d and dose response g were parameters in exactly this sense, and the whole of the Keeping a Model Current section is about how parameters move.

The step function f takes the state, the input and an interval h and returns the state one interval later:

$$x_{k+1} = f(x_k, u_k, \theta, h) \tag{6.1}$$

Equation 6.1 is the whole of what a simulator calls, in a loop, to produce a simulation. Chapter 3 said a model does not run and a simulation is one execution of it; the step function is the model's half of that bargain, and the loop is the simulator's.

The output function g reads from the state what a sensor would report:

$$z_k = g(x_k, \theta) \tag{6.2}$$

The output z in Equation 6.2 is a separate record from the state because a sensor rarely reads the state directly. An accelerometer on a tower reports an acceleration, not the tower's displacement; a thermocouple reports the temperature at one point of a gearbox, not the temperature of the oil as a whole. The output function is where the model says what its state would look like through the sensors Chapter 5 described, and it is the half the state and estimator chapter will lean on hardest: a residual is z predicted minus z measured.

Table 6.1 collects the five with the turbine's rotor as the instance, so that each has a concrete face before the sections that follow give it several.

Table 6.1: The five parts of a model, with the turbine's rotor as the instance. The right-hand column is the test that separates state from parameter: what changes it, and how often.

Part	Written as	For the turbine's rotor	Who changes it
state record	x	rotor speed	the step, every interval
input record	u	wind speed, generator command	the world, every interval
parameter record	θ	rotor inertia, blade geometry	an engineer, or an update, between runs

Part	Written as	For the turbine's rotor	Who changes it
step function	$x_{k+1} = f(x_k, u_k, \theta, h)$	new speed from torque imbalance and inertia	never during a run
output function	$z_k = g(x_k, \theta)$	the speed the SCADA counter reports	never during a run

6.2.2 Three notions of time, three signatures for the step

The interval h in the step function hides a choice. There are three ways a model can think about time, and they give the step three different signatures.

Continuous time. The model says how fast each state is changing right now – a **rate rule**, this chapter's plain word for what the mathematics appendix builds under *Quantities, Rates and Equations* and calls an ordinary differential equation (ODE). A rate rule has no step in it; a solver supplies one, and the appendix's *From an Equation to a Loop* shows Euler's step doing so: new value equals old value plus rate times step. The step function of a continuous-time model is therefore "call the solver for h seconds", and h is the solver's choice as much as the modeller's.

Sampled time. The model is written directly as a rule from this sample to the next, at a fixed interval – the difference equation of the appendix. Chapter 2's dose rule is one: it reads the moisture every T_c hours and never asks what happened in between. Here h is fixed by the sensor or the controller and the step function is exactly Equation 6.1 with h constant.

Event-driven time. Nothing happens until something happens. The model's step is "advance the clock to the time of the next event and apply it", and h is whatever gap the events leave, different every step. The appendix's *What a Solver and a Simulator Do* runs this step from a priority queue, and the Discrete-State and Discrete-Event Models section builds a model that lives in it.

The three are not rivals. A twin of a turbine holds a continuous-time model of the rotor, a sampled-time model of the ten-minute SCADA statistics and an event-driven model of the vessel that comes when a gearbox fails, and the simulators half of this part is largely about making the three tick together.

6.2.3 Deterministic or stochastic

A **deterministic** step returns one next state. A **stochastic** step returns a distribution over next states – or, in code, one draw from it, with the understanding that a different draw was possible. The difference is not in the physics but in what the model admits it does not know. A wearing bearing wears at a rate nobody can pin down to three decimal places, and a model that returns one wear level next month is overstating what it knows; a model that returns a spread is not. The Probabilistic Models and Bayesian Updating section is where the step starts returning distributions, and the fourth criterion below is why.

6.2.4 White, grey and black box

The literature sorts models by how much of the physics is written in them. A **white-box** model is written from physical laws – first principles such as Newton's second law, closed with constitutive relations for friction, damping and boundaries; a **black-box** model is fitted from measured data with no law inside; and a **grey-box** model sits between, with the physics guiding a structure whose parameters, or whose correction terms, are fitted from data [6]. The structural-health-monitoring literature reads the digital twin itself as the middle position: the model-based

approach is the white box, the vibration-based approach the black box, and the twin stands between them [40]. The first three model types in this chapter are white-box models that become grey the moment a parameter is fitted to this asset, which every one of them is. The black end of the spectrum – a network trained to map inputs to outputs – is parked for the AI chapter, and the Reduced-Order and Surrogate Models section says what a fitted function may and may not stand in for.

6.2.5 The four criteria

A model can be a good model and a bad model for a twin. What makes the difference is not accuracy alone but four properties, and every model type in this chapter is judged against the same four. They are the book's own framing, distilled from what the published twins in the sections below had to get right.

1. **Evaluable from this asset's state with today's readings.** The model's inputs must be things the twin actually has: the SCADA stream, the accelerometers, the bearing temperature Chapter 5 carried. A model that needs a wind field measured at fifty points along the blade, when the twin has one anemometer, is a good model of a turbine and a useless model for this twin.
2. **Updatable.** When the asset changes – wears, is repaired, is re-commissioned – the model must be able to follow. A model whose parameters are baked into a compiled binary nobody can rebuild fails this criterion however accurate it was on delivery.
3. **Inside the decision's time budget.** Chapter 1 priced each value pattern by the latency of its decision. A model that takes four hours to run cannot serve a decision due in ten minutes, and the appendix's *What a Solver and a Simulator Do* gives the one number, the real-time factor, that says whether it can.
4. **Honest about uncertainty.** The model must say how sure it is, so that a decision taken on its output can weigh the risk. A remaining life of "fourteen months" is less useful than "between nine and twenty months, most likely fourteen", and often less true.

The fourth is the one software engineers find strangest, because most functions we write return values, not distributions. It is also the one the last of the five model types exists to satisfy.

6.2.6 What the six value patterns ask of a model

Chapter 1's six value patterns – monitor, diagnose, predict, decide, control, certify – each stress a different subset of the four criteria, and Table 6.2 says which. Read it down a column to see which criterion is most often decisive: the time budget appears in every row from predict onward, which is why the two cheap model types carry most of the chapter's decisions.

Table 6.2: The six value patterns of Chapter 1 against the four criteria each stresses most. A model chosen for monitoring may fail the moment the twin is asked to predict.

Value pattern	What the model must do	Criteria stressed
Monitor	read the state as the sensors would: the output function	evaluable
Diagnose	predict what the readings should have been, to form a residual	evaluable, updatable

Value pattern	What the model must do	Criteria stressed
Predict	run forward from the estimated state, with a spread	time budget, honest about uncertainty
Decide	run forward many times, once per option	time budget, honest about uncertainty
Control	run forward inside the actuator's loop	time budget, evaluable, updatable
Certify	stand in for the asset with a stated error	honest about uncertainty, updatable

One reading of Table 6.2 to carry forward: the same twin usually climbs Chapter 1's ladder over its life, from monitoring to prediction to decision, and the model that served the first pattern on that ladder is rarely the one that serves the last. The Choosing a Model Type section returns to this, once the five types are on the table.

6.3 The Running Example: An Offshore Wind Turbine

One asset carries the five sections that follow, so that each model type is met as an answer to a question about the same machine rather than as a technique in the abstract. The turbine is the one Chapter 5 used for its worked stream, made concrete enough to model.

The machine. Three blades on a hub turn a low-speed shaft at around one revolution every four seconds in a good wind. A gearbox steps that up by a factor near a hundred to a generator, which the turbine's controller loads with a torque that sets how much power leaves. The nacelle holding all of this sits on a steel tower some eighty metres tall, and the tower stands on a foundation: a monopile driven into the seabed for a fixed turbine, or a floating spar or semi-submersible held by mooring lines for one in deep water. The two published twins this chapter leans on are both of floating turbines – a physics-based twin validated on a full-scale floating prototype that estimates the tower's loads and its fatigue life from a handful of standard measurements [164], and a diagnostic twin of an operational floating turbine that watches its gearbox temperatures for anomalies [22] – and the offshore-structures literature treats the tower and foundation as a structure whose measured dynamics should be compared with a predictive model [165]. Nothing in the sections below depends on whether the foundation floats; where it matters, the text says so.

What the operator sees. The turbine is in one of a handful of modes at any moment: idle below the cut-in wind speed, starting up, producing, curtailed when the grid operator asks for less than the wind would give, stopped on a fault, or under maintenance. Those six words are the whole of the turbine to the control room, and a section below models them as exactly that.

What happens when something breaks. Nobody drives to an offshore turbine. A crew-transfer vessel, or for a heavy component a jack-up vessel with a crane, is dispatched from port, and it sails only when the wave height and wind are below the vessel's limits, a **weather window** that in a North Sea winter may not open for days. A fault therefore costs the lost production from the moment of the fault to the end of the repair, and most of that time is spent waiting for a vessel and a window. That waiting is the thing a discrete-event model is for.

What the twin can read. Table 6.3 lists the sensors, at the rates Chapter 5 gave them. The supervisory control and data acquisition (SCADA) system reports wind speed, active and reactive power, rotor and generator speed, blade pitch and the temperatures of the shaft bearing, brakes, generator and gearbox oil [22]; on a commercial turbine those arrive as ten-minute statistics

[162]. A condition-monitoring system samples drivetrain vibration at hertz to tens of hertz. Tower accelerometers, and on a floater inclination and position sensors, sample at kilohertz, and the published structural twin needs only those plus the SCADA power, pitch and rotor speed to estimate the loads along the tower [164].

Table 6.3: What the turbine's twin can read, and which model each stream feeds. The rates are Chapter 5's; the right-hand column is this chapter's map from data to model.

Sensor	Reads	Rate	Feeds the model of
SCADA counters	power, rotor and generator speed, pitch	ten-minute statistics; hertz on a research turbine	rotor and drivetrain; operating mode
SCADA thermocouples	shaft bearing, brake, generator and gearbox oil temperatures	ten-minute statistics	gearbox oil; bearing wear
condition monitoring	drivetrain vibration	hertz to tens of hertz	bearing wear
tower accelerometers	tower acceleration at two or three heights	kilohertz	tower and blade structure; loads
floater sensors	inclination, position	hertz	structure, on a floating foundation
control room log	mode changes, alarms, vessel dispatches	on each event	operating mode; vessel dispatch

The decisions. Each model type below serves one decision, and Table 6.4 states them in advance so that each section can open on its question. The decisions are chosen so that no single model type could serve them all, which is the chapter's argument in one table: a twin of a real asset carries several models, and choosing among the types is choosing per decision, not per asset.

Table 6.4: The turbine's five decisions and the model type that serves each. The Choosing a Model Type section returns to this table with the reasons.

Decision	Question the operator asks	Model type that serves it
power and thermal expectation	is the rotor speeding up as it should, and will the gearbox oil overheat in the next ten minutes?	lumped first-principles
availability and repair planning	which mode is the turbine in, and when will a vessel get to it?	discrete-state and discrete-event
structural integrity	has the tower's stiffness changed since commissioning?	high-fidelity field
load monitoring every second	what load is the tower carrying right now, and how much fatigue did this gust cost?	reduced-order and surrogate

Decision	Question the operator asks	Model type that serves it
bearing replacement	how long before the main bearing must be replaced, with what confidence?	probabilistic, Bayesian updating

6.4 Lumped-Parameter First-Principles Models

Is the rotor speeding up as it should, and will the gearbox oil overheat in the next ten minutes? Both halves of that question are "how much" questions about a few quantities, and the model type that answers them is the oldest one: write down what the physics says about how each quantity changes, and step it forward.

6.4.1 Conservation is where rate rules come from

Every rate rule in this section, and most in engineering, comes from one sentence: **what flows in, minus what flows out, is what accumulates.** Energy into a gearbox minus energy out is energy stored in its oil, which is a temperature rise. Torque turning a rotor minus torque resisting it is stored angular momentum, which is a speed rise. Water into a pot minus water out is stored moisture, which was Chapter 2. The sentence is a rate rule as soon as "accumulates" is read as "the rate of change of the stored quantity is", and the mathematics appendix's *Quantities, Rates and Equations* is where that reading is built.

Lumped means the quantity is treated as one number for a whole component – one temperature for the gearbox oil, one speed for the rotor – rather than a value at every point. When a component's properties are close enough to uniform over its geometry, its parameters can be aggregated over the whole object and the object reduced to a point, and such lumped-parameter models are written as ordinary differential equations, or as differential-algebraic equations when constraints join them [166]. The High-Fidelity Field Models section is what happens when that assumption is dropped.

6.4.2 Three roles, four domains

Once the rate rules are written for a few systems a pattern shows. Every element plays one of three roles: it **stores** a quantity, it **dissipates** it, or it **supplies the effort** that drives it. A capacitor stores charge, a resistor dissipates it and a battery supplies the voltage; a mass stores momentum, friction dissipates it and a force drives it; a tank of oil stores heat, a cooler dissipates it and the gearbox's losses supply it. Table 6.5 lays the analogy out across the four domains a turbine's twin will meet. This is the observation the bond-graph formalism is built on – it unifies the physical domains at the lumped level by recognising the analogy between energy storage and dissipation across electrical, mechanical, thermal and hydraulic systems, and connects elements by the power that flows between them [166] – told here without its notation, because the observation is what a software engineer needs and the notation is not.

Table 6.5: Storage, dissipation and effort in four domains. Each row is one idea in four costumes; the last row is the rate rule each domain's conservation sentence becomes.

Role	Mechanical (rotation)	Electrical	Thermal	Hydraulic
stores	inertia holds angular momentum	capacitor holds charge	thermal mass holds heat	tank holds volume
dissipates	friction, drag	resistor	cooler, conduction to ambient	pipe friction, leak
supplies effort	applied torque	voltage source	heater, losses as heat	pump
the stored quantity's rate	torque imbalance over inertia	current over capacitance	net heat flow over thermal mass	net flow over tank area

The payoff of Table 6.5 is that a software engineer who has understood one column has understood the others. The turbine's rotor is the first column and its gearbox oil is the third, and the two rate rules below are the last row of each.

6.4.3 Equations, not assignments

A programmer's instinct is to write the rotor model as assignments: compute the torque, then the acceleration, then the new speed. A modeller in an equation-based language writes relations instead – torque equals inertia times acceleration – without saying which side is computed from which. This is the **acausal** style: a declarative style based on equations rather than assignments, in which which variables are inputs and which are outputs is left unspecified and is fixed only when causality is assigned before the model runs [166]. The point is reuse. A gearbox model written acausally can be driven from its input shaft or from its output shaft without being rewritten, because the equations do not know which end is upstream; an assignment-based model would need a second version.

The price is a translator. Before an acausal model can run, something has to sort its equations into an order in which each unknown is computed from things already known – which is exactly the assignment program the programmer would have written by hand, generated instead. Modelica is the language most twins in the literature use for this, and a Modelica model reaches a twin as a compiled unit: TwinOps, for instance, exports Modelica models as functional mock-up units, generates the interface and glue code for each, and co-simulates them with an architecture model while the same pipeline builds the real system [167]. Sorting happens at that compile step, and when it fails – because the equations do not determine the unknowns, or determine them twice – the failure is a compile error and not a wrong number at run time, which is the second reason engineers accept the translator.

Constraints are equations with no rate in them, holding at every instant beside the rate rules. The gear ratio is one: the generator's speed is the rotor's speed times the ratio, always, not "eventually". A published architecture for cyber-physical twins decomposes the system level by level from the whole to the components, writes each component's physics as ordinary differential equations and the connections between components as algebraic equations, and the model is the coupling of the two [168]. The appendix's *Stiff Equations for Digital Twins* is where the solver's

side of that coupling is discussed; here it is enough that a constraint is a relation the state must satisfy, and a translator uses it to eliminate a variable rather than to step one.

6.4.4 The turbine: six lines and a step function

Two states, two rate rules, one constraint, and a step. The symbols are in Table 6.6.

Table 6.6: The turbine's lumped model: two states, four inputs, four parameters and the step. The values are illustrative for a turbine of about two megawatts and are used in the worked step.

Symbol	Meaning	Unit	Role	Value used below
ω_{r}	rotor speed	radians per second	state	1.70
T_{oil}	gearbox oil temperature	degrees Celsius	state	48.0
Q_{a}	aerodynamic torque on the rotor	newton metres	input, from wind and pitch	1.50×10^{6}
Q_{g}	generator torque, referred to the rotor shaft	newton metres	input, from the controller	1.44×10^{6}
P_{loss}	gearbox loss power, as heat into the oil	watts	input, from the power flowing through	4.0×10^{4}
T_{amb}	nacelle air temperature	degrees Celsius	input	12.0
J	rotor and drivetrain inertia, referred to the rotor shaft	kilogram square metres	parameter	6.0×10^{6}
C_{th}	thermal capacity of the oil and gearbox casing	joules per kelvin	parameter	2.0×10^{6}
k_{th}	heat-loss coefficient from oil to nacelle air	watts per kelvin	parameter	1.0×10^{3}
n_{gear}	gear ratio	–	parameter	97
h	step size	seconds	the loop's	1

The rotor. Torque in minus torque out is stored angular momentum; angular momentum is inertia times speed; so the speed's rate is the torque imbalance over the inertia:

$$J \frac{d\omega_{\mathrm{r}}}{dt} = Q_{\mathrm{a}} - Q_{\mathrm{g}} \tag{6.3}$$

The oil. Heat in minus heat out is stored heat; stored heat is thermal capacity times temperature; heat out is the loss to the nacelle air, proportional to the temperature difference:

$$C_{\text{th}} \frac{dT_{\text{oil}}}{dt} = P_{\text{loss}} - k_{\text{th}} \left(T_{\text{oil}} - T_{\text{amb}}\right) \tag{6.4}$$

Equation 6.3 says that a rotor left alone keeps its speed, and Equation 6.4 that oil left alone settles where loss equals gain; the step below moves both by one second.

The constraint. The generator turns at the gear ratio times the rotor speed, $\omega_{\text{g}} = n_{\text{gear}}\,\omega_{\text{r}}$, and the generator torque the controller sets on the fast shaft appears at the rotor as Q_{g}, the fast-shaft torque times the ratio. Two algebraic lines, no rates.

The output function. The SCADA counter reports the rotor speed in revolutions per minute, $z_1 = 60\,\omega_{\text{r}}/(2\pi)$, and the oil thermocouple reports $z_2 = T_{\text{oil}}$. Two more lines, and the model is six lines long.

The step. Apply Euler's step of the appendix – new value equals old value plus rate times step – to each rate rule, with the values of Table 6.6:

```
def step(x, u, theta, h):
    omega_r, T_oil = x
    Q_a, Q_g, P_loss, T_amb = u
    J, C_th, k_th = theta
    d_omega = (Q_a - Q_g) / J                          # the rotor's rate
    d_T = (P_loss - k_th * (T_oil - T_amb)) / C_th     # the oil's rate
    return (omega_r + h * d_omega, T_oil + h * d_T)
```

Run it once by hand. The torque imbalance is $Q_{\text{a}} - Q_{\text{g}} = 6.0 \times 10^4$ newton metres, so the rotor's rate is $6.0 \times 10^4 / 6.0 \times 10^6 = 0.010$ radians per second per second, and after one second $\omega_{\text{r}} = 1.71$. The oil's excess over the air is $T_{\text{oil}} - T_{\text{amb}} = 36$ degrees, the loss is $k_{\text{th}} \times 36 = 3.6 \times 10^4$ watts against 4.0×10^4 watts coming in, so the oil warms at $4.0 \times 10^3 / 2.0 \times 10^6 = 0.002$ degrees per second and after one second $T_{\text{oil}} = 48.002$. The oil is heading for the temperature at which loss equals gain, $T_{\text{amb}} + P_{\text{loss}}/k_{\text{th}} = 52$ degrees, and at this rate it will take most of the next ten minutes to get close: that is the "will it overheat" half of the question answered, with numbers, from six lines.

Two things to notice, because they recur. First, the two states move on wildly different time scales – the rotor changes by a per cent in a second, the oil by a hundredth of a degree – and the step size was set by the fast one. The appendix's *Stiff Equations for Digital Twins* explains why that is forced and what a better solver does about it; the cost here is that the oil rule is stepped six hundred times per SCADA reading to advance a fraction of a degree. Second, the model predicts T_{oil}, and the SCADA system measures it: the difference between the two, once a reading arrives, is a residual, and a residual that grows is a bearing warming or a cooler failing. That is the diagnostic twin's whole method, and it is why the published diagnostic twin of a floating turbine chose the gearbox temperatures as the quantities to model: they change slowly, so they need only the low SCADA rates, and an abnormal temperature is a component misbehaving [22].

6.4.5 Where else this model type appears

The rotor-and-oil model is small. The same model type scales to a whole system by composition. The cyber-physical architecture cited above builds a lunar probe's power system as a library of lumped component models – environment, solar array, battery pack, controller, load, distribution – each a set of rate rules, joined by algebraic constraints into a system model that can be simulated at several levels of granularity [168]. In buildings, Spawn couples an equation-based

Modelica model of the heating, ventilation and control systems to EnergyPlus's envelope simulation, taking from the acausal side the flexibility to model novel heat-pump and storage systems that the sequential simulator's fixed loop topologies cannot express, and accepting in return longer translation and simulation times [169]. In both, the lumped model type is the frame that holds the rest together.

6.4.6 What it costs

Domain knowledge to write, and almost nothing to run. Writing the six lines above needed someone who knew that torque imbalance over inertia is an acceleration and that heat loss is proportional to a temperature difference; those are undergraduate physics, but they are physics, and a software engineer either learns them for the domain at hand or works with someone who has. Running them costs microseconds a step. The model passes the time-budget criterion for every value pattern in Table 6.2, and it passes the evaluability criterion because its inputs are the SCADA stream. What it does not do, alone, is say how sure it is: it returns numbers, not spreads.

6.4.7 How it is kept current

Two things drift, and they are kept current by two different mechanisms. The **state** – the rotor speed and the oil temperature the model thinks it has – drifts from the truth with every step, because the inputs are measured imperfectly and the model is not the turbine; it is corrected from every reading by an estimator, which is the state and estimator chapter's whole subject. The **parameters** – J, C_{th}, k_{th} – drift as the turbine ages, a fouled cooler lowering k_{th} over months, and they are refit from accumulated residuals on a slow schedule, which is the Keeping a Model Current section's subject. Nothing about the structure of the six lines changes until the turbine itself does.

6.4.8 One Euler step gives the state-space form

Close by writing the step once more, as a table. Stack the two states into a vector $x = \begin{bmatrix} \omega_{\text{r}} \\ T_{\text{oil}} \end{bmatrix}$ and the two driving quantities into $u = \begin{bmatrix} Q_{\text{a}} - Q_{\text{g}} \\ P_{\text{loss}} + k_{\text{th}}\, T_{\text{amb}} \end{bmatrix}$, the second entry being the heat that would flow in with the oil at nacelle temperature. Both rate rules are then weighted sums of the states and inputs, and one Euler step of both at once is:

$$x_{k+1} = F\,x_k + G\,u_k, \qquad F = \begin{bmatrix} 1 & 0 \\ 0 & 1 - h\,k_{\text{th}}/C_{\text{th}} \end{bmatrix}, \qquad G = \begin{bmatrix} h/J & 0 \\ 0 & h/C_{\text{th}} \end{bmatrix} \tag{6.5}$$

Equation 6.5 is the discrete-time state-space form the appendix builds under *Many Quantities at Once* and the state and estimator chapter runs its filter on. The turbine's F has a one in the corner because the rotor speed, left alone, keeps its value, and $1 - h\,k_{\text{th}}/C_{\text{th}} = 0.9995$ in the other because the oil's excess decays half a thousandth per second; multiply x by F, add G times u, and you have the two numbers computed by hand above. That a six-line physics model and a two-by-two table are the same model, written twice, is the fact this section hands on: the physics gives the twin its expectation, and the tables are the form in which the next chapter corrects it.

6.5 Discrete-State and Discrete-Event Models

Which mode is the turbine in, and when will a vessel get to it? Neither half of that question is "how much". The first asks which of six words describes the turbine; the second asks when a sequence

of things – a fault, a dispatch, a weather window, a sailing, a repair – will have happened. Both are questions about discrete states and the events that move between them, and this is the model type for them.

6.5.1 Home turf: a finite state machine with guards and actions

A software engineer has written this model many times without calling it one. A **finite state machine** is a finite set of modes, exactly one of them active, and a set of **transitions**, each fired by an event, each with a **guard** – a condition on the state or the inputs that must hold for the transition to fire – and an **action**, what is done on firing. Figure 6.2 draws the turbine's operating modes as one. Idle becomes start-up when the wind rises above the cut-in speed; start-up becomes producing when the rotor reaches its rated speed; producing becomes curtailed on a grid command and returns when the command is lifted; any operating mode becomes fault on an alarm; fault becomes maintenance when the vessel arrives; maintenance becomes idle when the repair is done.

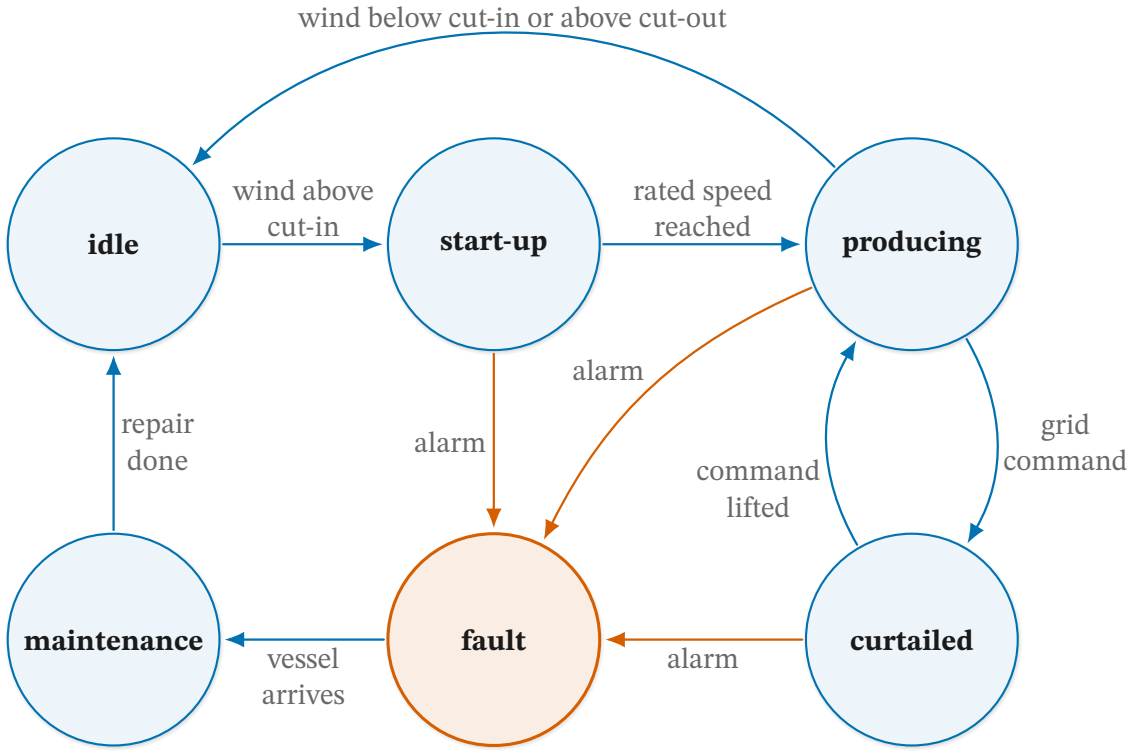


Figure 6.2: The turbine's operating modes as a finite state machine. Each arrow carries its guard; the actions – log the change, set the generator torque to zero, raise a work order – are not drawn. Alarms lead to the fault mode from every operating mode.

In the terms of the What Every Model Has section: the state record is one value from an enumeration of six; the inputs are the wind speed, the grid command, the alarm flags and the vessel's arrival; there are no parameters yet; the step function is "if the guard of any outgoing transition holds, move and perform its action"; and the output function is the mode itself, which is what the control room log records. The state is one word, which is why this model type answers "which" and not "how much".

6.5.2 Adding time, twice

A plain state machine has no clock. It moves when an event arrives and otherwise waits, and it cannot say how long start-up takes. Time enters in two steps.

Timed transitions. Give a transition a duration: start-up takes ninety seconds; a repair takes eight hours. The step function now has a clock, and a transition fires when its duration has elapsed as well as when its guard holds. That is enough for a single machine.

Timestamped events in a queue. The moment there are two machines – the turbine and the vessel – or one machine and a world that sends things at its own pace, the clean way to run them is to put every pending event on a **priority queue** ordered by its timestamp, pop the earliest, advance the clock to it, apply it, and let the application schedule new events onto the queue. This is the **discrete-event** abstraction: the step is "pop, advance, apply", and the state of the whole model is the modes of its machines plus the queue. The appendix's *What a Solver and a Simulator Do* runs exactly this loop under *An event scheduler runs the discrete version*. It is also the view the discrete-event system specification (DEVS) formalises: each atomic model has a time-advance function that says how long it will stay in its state if nothing arrives, an internal transition for when that time runs out and an external transition for when an input arrives first, with the elapsed time since the last transition available so that an interrupted process can resume with the time it has left [166]. A twinning system's design passes through a stage of choosing such formalisms – differential equations, DEVS or Petri nets – to match its goals, before deployment [12]; the choice this section teaches is the second of those, with the third next.

6.5.3 Petri nets: the thing state machines do badly

A state machine has one active mode. Two vessels, four turbines, one spare gearbox and two weather windows have no single mode; they have counts, and the question is which activities can proceed at once and what they compete for. A **Petri net** answers it. Its state is a marking: a count of **tokens** in each **place**, where a place may stand for a process step ("waiting for a vessel") or a resource ("spare gearboxes in port") alike; a **transition** fires when every place feeding it holds enough tokens, consuming them and producing tokens in the places it feeds. Petri nets treat processes and resources as the same kind of thing, encode the state as a vector of natural numbers, and are built for concurrency and synchronisation – and, because they cover all possible behaviours rather than one run, they are used mostly for analysing properties, such as showing that two activities can never both hold the same crane, rather than for simulation [166]. A repair that needs a vessel *and* a spare *and* a window is three tokens consumed by one transition, and a state machine would need a mode for every combination.

Queues are the special case a software engineer meets most: things arrive, wait in a line, are served one at a time. Turbines with faults are the arrivals, the vessel is the server, and the waiting line is the lost production. The formalisms' own literature lists suitability for competition over shared resources – which is what leads to queueing – among the properties a modelling language is chosen for [166]. Any queue can be drawn as a small Petri net, and most are simulated as a discrete-event model instead.

6.5.4 The turbine: vessel dispatch with weather windows

Two machines and a queue. The turbine's mode machine is Figure 6.2. The vessel's machine has four modes: in port, waiting for a window, sailing, on site. The parameters this model type does have are durations and frequencies: the sailing time, the repair time, and the weather – modelled here as windows that open and close as events, with a mean gap between them that is the one number a planner most wants to know in winter. Table 6.7 pops the first five events from the queue for one fault.

Table 6.7: Five events popped from the queue for one gearbox fault, with the mode changes each causes. The waiting for a window, nine and a half hours here, is the part of the twenty hours no repair crew can shorten.

Clock	Event popped	Guard checked	Action, and events scheduled
02:10	gearbox alarm on turbine 7	turbine 7 is producing	turbine 7 to fault; generator torque to zero; schedule “dispatch requested” now
02:10	dispatch requested	vessel in port; a spare gearbox in port	vessel to waiting for a window; ask the weather model for the next window
11:40	weather window opens	vessel waiting	vessel to sailing; schedule “on site” at 11:40 plus the three-hour sailing time
14:40	vessel on site	window still open	turbine 7 to maintenance; schedule “repair done” at 14:40 plus eight hours
22:40	repair done	–	turbine 7 to idle; vessel to sailing home; log twenty hours and thirty minutes of lost production

Read Table 6.7 down the clock column: nine and a half of the twenty hours are the vessel waiting in port, and that is the number the model exists to produce. Run the same queue a thousand times with the weather windows drawn from their winter distribution and the lost-production hours per fault come out as a spread, which is the “when will a vessel get to it” question answered honestly. Run it with a second vessel, or a spare gearbox kept on the turbine, and the spread moves: that is the decision the operator is paying for.

6.5.5 Where else this model type appears

The manufacturing-twin literature’s own table of modelling techniques lists, under machine and process states, the finite state machine, the hybrid automaton and the hidden Markov model, with a hidden Markov model used for the reactive control of a robot cell’s components and a generic hybrid automaton written for the continuous and discrete dynamics of additive manufacturing [91]. In production, discrete-event simulation models abstract a group of machines into event-driven operations and can form a twin of the whole production process [49]; and the software engineering literature on twins records the same abstraction carried into a hospital, where a discrete-event simulation integrated with pervasive sensing forms the hospital’s twin [170]. Patients, parts and turbines with faults are the same arrivals to the same queue.

Hybrid systems are where this model type meets the previous one. A turbine in start-up obeys one set of rate rules, a turbine producing under torque control another, and a turbine in fault with the brake on a third: the mode machine chooses which rate rules run, and the rate rules’ states decide when a guard fires. A state machine whose modes each carry their own rate rules is a **hybrid automaton**, and the hybrid automaton of an additive process cited above is one. The simulators half of this part takes up how the two clocks are kept in step; here, the design point is that the mode is state too, and belongs in the state record beside the speed and the temperature.

6.5.6 What it costs

Almost nothing to run: popping events from a queue is cheap, and a year of a turbine's mode changes is a few thousand events. The hard part is not computing but deciding. Which events matter? A model that includes every alarm code the controller can raise has a state machine nobody can read and durations nobody can estimate; a model that collapses them to "fault" answers the availability question and no other. The cost is the modeller's judgement about the decision the model serves, and it is paid once, up front.

6.5.7 How it is kept current

Its parameters are durations and frequencies – how long a start-up takes, how often a window opens, what fraction of faults need a jack-up vessel – and every one of them is re-estimated from the event log the control room already keeps. This is the easiest model type to keep current, because the log *is* the calibration data, and the Keeping a Model Current section's slow schedule can be "recompute the means from the last two winters". The structure – which modes exist, which transitions join them – changes only when the turbine or the operating rules do, and that is configuration's business.

6.6 High-Fidelity Field Models

Has the tower's stiffness changed since commissioning? A crack at a weld, a scoured seabed around a monopile, a loosened flange: each changes how the tower bends, and none of them is a lumped quantity. The question is about the tower as a body, and the model type that answers it treats it as one.

6.6.1 From a handful of states to a field

The previous two model types kept one number per component. A **field** model keeps a value at every point of the component: the stress at every point of the tower wall, the temperature at every point of a gearbox casing, the air speed at every point around a blade. The state is no longer a record of a few quantities but a function over space, and a computer cannot hold a function over space. So the body is cut into small cells – a **mesh** – and the field is stored at the cells' corners, the **nodes**, with a simple rule for how it varies inside each cell, the **element**. Figure 6.3 shows the two views of the same tower section side by side.

Now the thing to see. Once the mesh is drawn, every node's displacement is a state, and the physics at each node – force in minus force out equals mass times acceleration, the same conservation sentence as the previous section – is a rate rule relating it to its neighbours. A field model is therefore a lumped model with a million states, written automatically by a program that walks the mesh, rather than by hand. It has the same anatomy as everything in this chapter: a state vector, an input (the wind and wave loads at the nodes on the surface), parameters (the steel's stiffness and density, the wall thickness, the soil's stiffness at the base), a step function and an output function that reads the acceleration at the nodes where the accelerometers are. It is the size that changes, and with it the cost and what can be computed.

Modes are computed, not assumed. The appendix's *Many Quantities at Once* introduced a mode as a shape a structure prefers to move in, at a natural frequency, and gave the two-mass example whose frequencies followed from two stiffnesses somebody had to supply. A field model supplies them. From the mesh and the parameters it computes the tower's mode shapes and natural frequencies – the first sway, the second sway, the first twist – without anybody assuming a stiffness for the whole tower, because the stiffness is a consequence of the wall thickness and



(a) three lumps: three states
stiffness and mass assumed

(b) a mesh: every node a state
modes computed, not assumed

Figure 6.3: The same tower section as a lumped model and as a field model. In panel (a) three masses on two springs give three states and a stiffness somebody assumed; in panel (b) a mesh gives a state at every node and a stiffness computed from the wall thickness and the steel.

the material, element by element. That is what makes the field model the right tool for the stiffness question: a change in stiffness anywhere moves a natural frequency somewhere, and a natural frequency is something an accelerometer can measure.

6.6.2 Two families

Finite-element models are the field models of solids and structures: towers, blades, gearbox casings, bridge decks. The finite element method (FEM) is the mesh-walking program described above, and its output is displacements, stresses and modes. **Computational fluid dynamics** (CFD) is the field model of fluids: the air around a blade, the water around a foundation, the hot gas in a furnace. The same idea – mesh, conservation at every cell, a huge state vector – with the complication that the fluid moves through the mesh, so the equations at each cell involve what is flowing in from upstream. The two are often coupled, because the fluid's pressure is the structure's load, and that coupling is one of the reasons these models are expensive.

6.6.3 The turbine: computed frequencies against measured ones

The tower and blades are meshed; the mesh, the steel's properties, the wall thicknesses and an assumed soil or mooring stiffness are the parameters; and the model computes the first few natural frequencies. Meanwhile the accelerometers of Table 6.3 record the tower's motion, and a procedure the appendix names – operational modal analysis, which extracts natural frequencies and mode shapes from vibration records without knowing the loads – produces the measured frequencies. The comparison is the decision: if the computed first sway frequency was 0.31 hertz at commissioning and the measured one is 0.31 hertz today, the tower's stiffness has not changed; if the measured one has drifted to 0.29, something has. The offshore twin literature describes exactly this loop for North Sea platforms, in five levels from screening and diagnostics through finite-element model updating and wave-load calibration to quantified uncertainty and accumulated fatigue [40], and lists the model parameters that move the frequencies – mass and centre of gravity, element stiffness, local joint stiffness, soil stiffness and damping – as the ones an update adjusts [165]. The diagnostic twin of the floating turbine sits on the same asset and reads the SCADA side of it [22]; the field model is what would read the structural side.

6.6.4 Where else this model type appears

The structural twin of a small unmanned aircraft is a finite-element model of its wing, calibrated so that the first two bending modes' natural frequencies match the ones extracted from strain measurements after an initial-condition test; once calibrated, the model computes the higher bending, torsional and skin-buckling modes that are hard to measure, and its structural-health parameters are percentage reductions in stiffness in regions of the wing – the aircraft's version of the tower question [171]. In construction, a twinning platform generates building twins automatically from the building information model, exchanged as Industry Foundation Classes, which carries the geometry a field model of the building starts from, so that the twin is instantiated from the building's own design documents rather than from a separate modelling effort [37]. Both are the same move as the turbine: the mesh comes from the design, and the parameters are what the twin has left to learn.

6.6.5 What it costs

Hours per run, specialist tooling, and somebody who knows how to mesh. A tower model of a few hundred thousand nodes takes minutes to compute its modes and hours to simulate a storm; a blade in CFD takes days. The tooling is a commercial finite-element or CFD package or a research code, and the meshing is a skill of its own – too coarse and the model misses the stress concentration at the weld, too fine and the run does not finish. The manufacturing-twin review reaches the conclusion this cost forces: high-fidelity numerical models are well suited to twin applications in principle, but may not be appropriate in practice because fidelity comes at the price of computational cost and simulation time, and the remedy the literature reaches for is reduced- order modelling or simplification [91]. Against the four criteria, the field model fails the time budget for every value pattern below certify, and the next section is how it buys that back. It passes the others: its inputs are the loads and the accelerometers, its parameters are updatable, and once its uncertainty is quantified it is the most honest model of the tower there is.

6.6.6 How it is kept current

By **model updating**: adjusting the model's parameters until the frequencies it computes match the frequencies the accelerometers measure. The published twin-platform case does this for a steel beam – vibration records are processed by operational modal analysis to extract the natural frequencies, a detailed finite-element model is built, and the measured frequencies become the targets of an optimisation over the model's design variables, which is finite-element model updating [172]. The turbine's version adjusts the soil or mooring stiffness and the local joint stiffnesses until the computed sway frequencies match the measured ones, and records how far they had to move, which is the stiffness question's answer. It is done on a slow schedule – after commissioning, after a storm, once a season – because each update is hours of compute; the Keeping a Model Current section places it on that schedule beside the lumped model's parameter refit.

6.7 Reduced-Order and Surrogate Models

What load is the tower carrying right now, and how much fatigue did this gust cost? The field model of the previous section knows the answer, and takes hours to say it. The decision wants it every second. There are two ways to buy that time back, and they are different enough to deserve different names.

6.7.1 Reduce: keep a few combinations of the million states

The field model's million nodal displacements are not a million independent things. The appendix's *Many Quantities at Once* made the point for a two-mass system: any motion is a mix of the modes, and a handful of modes describes most of what a large linear system does. A tower sways mostly in its first and second sway modes and twists a little in its first twist mode; the other 999,997 combinations of its nodal displacements carry almost none of the motion. A **reduced-order model** keeps only the few combinations that matter – a **reduced basis**, typically the leading modes – and writes the rate rules for their amplitudes instead of for the nodes. The state vector shrinks from a million to five, the step function is a five-by-five table, and the loop runs in microseconds. The full field can be reconstructed at any moment by mixing the five mode shapes in the current amplitudes, which is how a five-state model still answers "what is the stress at the weld".

The model produced this way is a lumped model again, and everything the Lumped-Parameter First-Principles Models section said about steps, solvers and the state-space form applies to it unchanged. What is new is where it came from: its five states were chosen by the field model's modes, not by an engineer's judgement, and its parameters were computed, not assumed.

6.7.2 Replace: fit a cheap function from inputs to outputs

The second way does not shrink the model; it stops running it. If the decision needs one number – the fatigue damage a ten-minute interval of given wind speed, wave height and turbulence does to the tower – run the field model a few hundred times offline over the range of conditions the turbine will meet, record the input-output pairs, and fit a cheap function through them. That function is a **surrogate**. It can be a polynomial in the inputs, a **response surface**; a **Gaussian process**, which fits a distribution over functions and so returns a spread as well as a value; or a network. The twin literature describes surrogates in exactly these terms – response surfaces, kriging, support vector machines and neural networks that mimic computationally expensive physics models – and notes that they are built from training data taken from the original simulation, not from the asset [92]. Once fitted, a surrogate is evaluated in microseconds, and the big model is never run again in operation.

6.7.3 The turbine: a reduced tower and a fatigue surrogate

The reduced tower. Project the finite-element tower onto its first three sway and twist modes. Feed the three-state model the tower accelerations and the SCADA power, pitch and rotor speed, and it estimates the load at every section of the tower every second – which is the load-monitoring decision. The published relative of this is the executable-twin literature's wind-turbine blade: a detailed finite-element model of the blade compressed into a reduced-order model and paired with a state estimator gave a virtual sensor that shows the full strain field of the blade in real time on a test rig instrumented with only a few strain gauges [20]. The same paper did the same for a truck anchorage whose critical locations could not be instrumented at all, which is the general point: a reduced model plus a few sensors reads what no sensor can reach. The state estimator that makes it work is the state and estimator chapter's; the reduced model is what it runs on.

The fatigue surrogate. Fatigue is the slow accumulation of damage from many small load cycles, and computing it from a load history is a standard but expensive calculation. Run the field model, or the reduced one, over a grid of wind speeds, wave heights and turbulence intensities; compute the ten-minute fatigue damage for each; fit a Gaussian process to the table. In operation, the twin reads the ten-minute SCADA statistics, evaluates the surrogate, and adds the result to the tower's running fatigue total, with a spread from the Gaussian process that says

how well the surrogate knows this corner of the input space. That is "how much fatigue did this gust cost", every ten minutes, for microseconds.

6.7.4 Where else this model type appears

The aircraft twin above is built on a **library** of component-based reduced-order models – reduced models of the wing's components, built component by component – so that any asset state the library covers can be simulated in real time, and a classifier reads the sensor data and picks which library models best describe this aircraft now; sharing the library across many aircraft is what lets one modelling effort serve many twins [66]. In manufacturing, the argument has been made for reduced-order models that are interpretable, so that an operator can see what a reduced state means, and reusable across processes in a manufactory-as-a-service setting [173]. And a surrogate can sit inside a probabilistic update rather than replace one: a multi-fidelity neural network trained on a finite-element model of several thousand degrees of freedom maps damage and operating parameters to sensor readings, and a Markov chain Monte Carlo procedure calls it thousands of times to update the distribution of a structure's damage state from noisy observations – a use that would be impossible with the full model in the loop [174]. That last case is the bridge to the next section.

6.7.5 A surrogate mimics a computational model, not this asset

The first of two warnings, and the one the literature is most emphatic about. A surrogate is fitted to input-output pairs from a *model*. It knows what the finite-element tower does; it knows nothing about turbine seven. Its confusion with a twin comes from a surface similarity – both use data to update what they know – but a surrogate alone is not a twin, because it maintains no representation of the state of an instance of a physical system; it mimics a computational model, and belongs *inside* a twin, where the original model is too slow for the decision interval, not in place of one [92]. In this chapter's terms: the surrogate is a fast output function for a model whose state and parameters the twin still has to hold and update. Drop the state and the update and you have a fast calculator, not a twin.

6.7.6 A surrogate is valid only where it was trained

The second warning. A response surface fitted over wind speeds from four to twenty-five metres per second says nothing about a thirty-metre-per- second storm; it will return a number, and the number will be confidently wrong. The Gaussian process is the honest choice among surrogates for exactly this reason – its spread grows away from the training points, so it says "I do not know" where a polynomial would not – but even it only knows that it does not know, not what the answer is. Every surrogate needs a check that its inputs are inside the region it was fitted on, and a fall-back – the reduced model, or the field model on a slower schedule – for when they are not.

6.7.7 What it costs

The big model must exist first. A reduced model is a projection of a field model and a surrogate is a fit to one; neither can be built without the hours of meshing and running the previous section priced. What they buy is the time budget: after the offline cost is paid, both run in microseconds and pass criterion three for every value pattern. Against the other criteria, the reduced model is as evaluable and updatable as the field model it came from, and honest about uncertainty only if the state estimator makes it so; the surrogate is honest only if it is a Gaussian process or carries an explicit validity check.

6.7.8 How it is kept current

When the field model is updated – the tower's stiffness re-fitted after a storm, as in the previous section – the reduced basis and the surrogate's training table are both stale, and both are rebuilt from the updated model, or the surrogate is refit on a fresh set of runs. That rebuild is hours, on the field model's schedule, not seconds; the twin runs the old reduced model until the new one is ready, and should know that it is doing so. The National Academies' report on research gaps for digital twins lists this as open: dynamic updating and adaptation must often happen on the fly under computational and time constraints, surrogates are what make such updating feasible, and yet the surrogates must themselves be updated and re-validated as the twin's virtual representation evolves [90]. A twin that cannot say which version of its surrogate produced a number fails the audit the Keeping a Model Current section asks for.

6.8 Probabilistic Models and Bayesian Updating

How long before the main bearing must be replaced, and with what confidence? The second half of that question is the point. The four model types so far return numbers, and a number for a bearing's remaining life is a guess wearing a suit. This model type returns a distribution, and it is the one that makes the fourth criterion – honest about uncertainty – structural rather than an afterthought.

6.8.1 Probabilities on the arrows: the Markov chain

Start where the Discrete-State and Discrete-Event Models section left off. Put a probability on every arrow of the turbine's mode machine – from producing, the chance of a fault in the next hour is 0.002, of a curtailment 0.05, of staying put the rest – and the model's state stops being one word. It becomes a **belief**: a row of six probabilities, one per mode, that sums to one. Each step multiplies that row by the table of arrow probabilities to get next hour's belief. This is the **Markov chain** the appendix builds under *State Machines with Probabilities*, and it is worth one paragraph here and no more: it is the smallest stochastic model there is, its step returns a distribution because its state *is* a distribution, and the state and estimator chapter is where a noisy reading is folded into the belief. Here the chain is the bridge from "which mode" to "how sure".

6.8.2 Wear as a random walk with drift

A bearing does not have modes; it has a wear level that only rises. Call it W, measured as the vibration level the condition-monitoring system reports, in millimetres per second. The simplest honest model of it says two things: on average it rises at a **drift** μ per month, and month to month it wanders around that average by an amount with spread σ. Written as a step:

$$W(t+h) = W(t) + \mu\, h + \sigma\sqrt{h}\,\xi, \qquad \xi \sim \mathcal{N}(0,1) \tag{6.6}$$

Equation 6.6 is a **Wiener process** with drift, and the one unfamiliar thing in it is the square root. The random part scales with the square root of the interval, not the interval, because independent random kicks add up in variance, not in size: two months of wandering is not twice as far as one, on average, but about 1.4 times as far. Everything else is Euler's step with a random term. The degradation literature that twins draw on models wear this way – as a stochastic process, with the Wiener, gamma, inverse-Gaussian and Markov processes the common choices when the physics of the damage is not known well enough to write a rate rule for it [27].

Remaining useful life is then a distribution, not a date. Fix a failure threshold W_{f}, the vibration level at which the bearing must be replaced. The remaining useful life (RUL) is the time until W first crosses W_{f}, and because each run of Equation 6.6 crosses at a different time, the RUL is the spread of those crossing times. Its centre is roughly the distance to the threshold over the drift, $(W_{\mathrm{f}} - W)/\mu$; its width grows with σ and shrinks as the threshold nears. The predictive-maintenance twin literature defines RUL prediction as estimating the time to a component's failure from the system's data and lists the Wiener process first among its non-deep-learning methods [26]. A twin that reports "replace in fourteen months" has thrown away the width; one that reports "between nine and twenty, most likely fourteen" has kept it.

6.8.3 Hidden health states, noisy readings

The vibration level is not the bearing's health; it is a reading that depends on it, with noise from the load, the speed and the sensor. The model's true state – the wear – is **hidden**, and the twin sees it only through $z = W +$ noise. The appendix's *A chain seen through a noisy sensor* is this situation for a Markov chain, the hidden Markov model, and the state and estimator chapter is the machinery for estimating the hidden state from the readings. This section needs only the idea: the reading updates a belief about something unseen, and the next paragraph does that update for one parameter by hand.

6.8.4 Bayes' rule on one parameter, three candidates

Recall Bayes' rule from the appendix's *Probability and Distributions*: the belief in a hypothesis after a reading is proportional to the belief before it times how likely the reading was if the hypothesis were true. Prior times likelihood, normalised, gives posterior. Apply it to the drift μ, which is the parameter the RUL depends on most and the one nobody knows for this bearing.

Suppose fleet history says bearings of this type wear at one of three rates – slow, normal, fast – with μ of 0.5, 1.0 or 2.0 millimetres per second per month, and that half of bearings are slow, three in ten normal, two in ten fast. That is the prior. Last month the vibration level rose by 1.2 millimetres per second. Taking the month-to-month scatter of a one-month rise, sensor noise included, as a Gaussian with standard deviation 0.5, the likelihood of a 1.2 rise under each candidate is the Gaussian density at 1.2 with mean μ and standard deviation 0.5. Table 6.8 does the arithmetic.

Table 6.8: Bayes' rule on the bearing's drift, with three candidate values and one month's reading. The posterior column is the fourth column divided by its sum; one reading has moved the belief from "probably slow" to "probably normal".

Candidate μ	Prior	Likelihood of a 1.2 rise	Prior times likelihood	Posterior
0.5 (slow)	0.50	0.299	0.150	0.36
1.0 (normal)	0.30	0.737	0.221	0.53
2.0 (fast)	0.20	0.222	0.044	0.11
sum	1.00	–	0.415	1.00

Read Table 6.8 across the normal row: the prior 0.30 times the likelihood 0.737 gives 0.221, and dividing by the column sum 0.415 gives a posterior of 0.53. The reading was most consistent with normal wear, so normal wear gained belief; it was least consistent with fast wear, which

lost most. Next month's reading multiplies the posterior by a new likelihood and normalises again, and after a few months the belief will sit almost entirely on one row. Three candidates is a teaching simplification – a real twin carries μ as a continuous distribution – but the mechanism is identical, and the appendix's worked examples do it twice on the same bearing.

6.8.5 Model-form discrepancy, in words

Fitting μ assumes the Wiener model is right and only its parameter is unknown. It is not right; no model is. The bearing's wear may accelerate near the end, or respond to load in a way a constant drift cannot follow. The honest move is to admit that the model is wrong by an unknown amount, assume that amount is smooth – it does not jump about from month to month – and learn it from the residuals beside the parameters. The twin literature calls that amount **model-form uncertainty** or **model discrepancy**, and the standard way to learn it beside the parameters is the Kennedy and O'Hagan framework, in which the uncertain parameters are calibrated concurrently with a discrepancy term, which has been shown to improve prediction over calibrating the parameters alone [11]. In this chapter's terms: the model's output gains a second uncertain piece, and the update that moved μ moves it too. The credibility chapter is where the size of that piece becomes evidence for or against believing the twin.

6.8.6 The twin as a chain of belief updates

Step back and the whole twin has the shape this section has been building one piece at a time. Figure 6.4 draws it as a **probabilistic graphical model** (PGM): boxes for uncertain quantities, arrows for what depends on what. Along the top, the physical asset's true state moves forward in time on its own. Along the bottom, the **digital state** – the model's parameters and hidden states, held as distributions: the belief over modes, the wear level, the drift μ, the stiffness reduction in a tower section – moves forward too. Readings climb from the asset into the digital state and update it by Bayes' rule, exactly as Table 6.8 did; decisions computed from the digital state descend onto the asset. The figure is to be read, not derived: the published version defines the asset and its twin as two coupled dynamical systems, the upper path the physical state and the lower the digital state, coupled by observational data assimilated to update the digital state and by control inputs chosen from it [171], and the civil-structures twin built on the same idea encodes the repeating observations-to-decisions flow as a dynamic Bayesian network whose updated digital state drives prediction and maintenance planning [175].

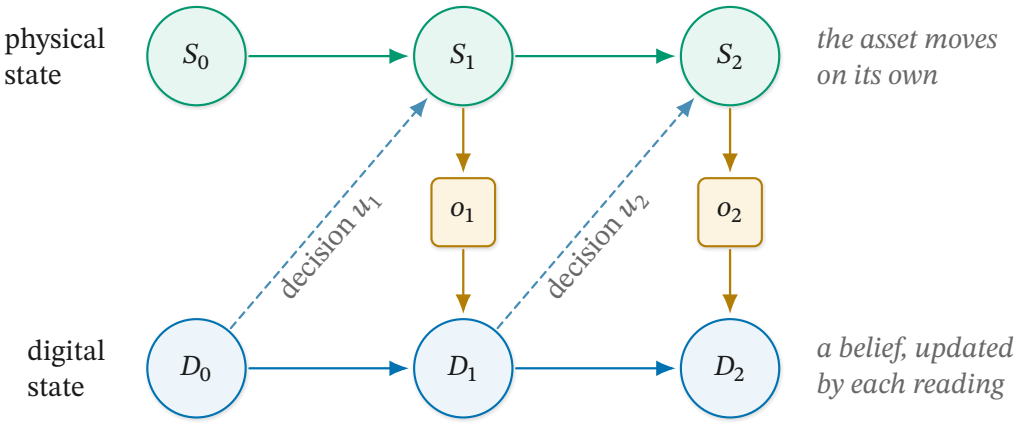


Figure 6.4: The twin as a probabilistic graphical model. The physical state advances along the top and the digital state along the bottom; each reading climbs into the digital state and updates it, and each decision descends from the digital state onto the asset.

The reason to draw Figure 6.4 here rather than in the architecture chapter is what it says about models. Every model type in this chapter is a way of writing the arrow along the bottom row – how the digital state moves from one step to the next – and every one of them is improved by being read as this section reads the Wiener model: with its parameters as beliefs, updated by readings. That reading is what the phrase "digital state" means in the literature: the set of variables that characterise a twin's models, often not directly measurable, estimated through noisy readings [27].

6.8.7 The turbine: a remaining-life distribution, not a date

Put the pieces together for the main bearing. The state is the wear level W, read through the vibration trend; the parameters are the drift μ and spread σ; the step is Equation 6.6; the update is Table 6.8, once a month. With the current level at 3.5 and the threshold W_{f} at 7.1 millimetres per second, the distance to failure is 3.6, and the posterior of the table says: with probability 0.36 the drift is 0.5 and the bearing has about seven months; with 0.53 it is 1.0 and about three and a half months; with 0.11 it is 2.0 and under two months. The RUL is that mixture, widened further by σ. The operator does not get a date. The operator gets "a one-in-ten chance this needs a vessel inside two months", which is the number a maintenance planner can act on – book a provisional weather window, or not – and the number no deterministic model could have produced.

6.8.8 What it costs

Where the parameter has three candidates, the update is a table. Where it is a continuous distribution over several parameters and the model is nonlinear, the posterior has no closed form and is drawn by **sampling**: Markov chain Monte Carlo, Gibbs sampling and slice sampling are the names the twin literature gives for computing the posterior of uncertain parameters, and they are costly enough that surrogates are routinely placed inside them [11]. This chapter names them and teaches none; the cost to know is that a Bayesian update of a rich model is thousands of model evaluations, which is why it is done on the slow schedule and why the Reduced-Order and Surrogate Models section matters to this one. Against the four criteria, this model type is the one built to pass the fourth, and it inherits the other three from whichever model it wraps.

6.8.9 How it is kept current

This model type *is* updating. Its parameters are beliefs and every reading moves them; there is no separate "keep current" step because the step function and the update are the same machinery seen from two sides. What the next section adds is the generalisation: the other four model types can be kept current the same way, on three different clocks.

6.9 Keeping a Model Current

Every model type above closed on the same sub-heading, and the answers had a shape. Something changed every reading, something changed on a slow schedule, and something changed only when the turbine itself did. This section names the three, says which chapter owns each, and gives the middle one – the one this chapter owns – its vocabulary.

6.9.1 Three rates, three homes

Table 6.9 is the section in one table. The **state** changes with every reading and is corrected by an estimator, which is the state and estimator chapter's subject; the **parameters** change on a slow

schedule or on a trigger and are re-estimated, which is this section's; the **structure** – which models exist, which states they carry, how they connect – changes when the asset changes and is declared in the twin's configuration, which the configuration and manager chapter owns.

Table 6.9: The three rates at which a twin's model changes, with the mechanism and the owner of each. A change applied at the wrong rate is the commonest way a twin drifts from its asset.

What changes	How often	Mechanism	Turbine instance	Owned by
state x	every reading: seconds to minutes	estimation: predict with the step, correct with the reading	rotor speed, oil temperature, wear level, mode belief	the state and estimator chapter
parameters θ	slow schedule, or on a trigger: weeks to seasons	re-estimation from accumulated residuals, or a Bayesian update	k_{th} after cooler fouling; μ each month; soil stiffness after a storm	this chapter
structure	when the asset changes: years, or on an event	reconfiguration: a new model, a new state, a new connection, declared	a replaced gearbox with a different ratio; a new sensor; a repowered turbine	the configuration and manager chapter

The last column of Table 6.9 is a rule about responsibility, and the middle column is why it matters. An estimator that is allowed to move a parameter every reading will "explain" a faulty sensor by re-fitting the inertia; a parameter refit that changes the structure will silently re-map a replaced sensor onto the old model's state. The published approaches to the third row keep it apart deliberately: one twin architecture runs two feedback loops, a behavioural loop that adapts the running model's settings and an architectural loop above it that reconfigures the first when the asset moves to a new lifecycle stage declared in an asset model [54]; another observes that a twin's schema must change whenever the physical system evolves, and that schema, instances and stored time series then have to be evolved together, which the platforms surveyed did not support [176]. Both are the configuration chapter's material; here they fix the boundary of this one.

6.9.2 Verification, calibration, validation

Three words the literature uses about a model's fitness, and they are three different questions, worth keeping apart because a positive answer to one is routinely mistaken for a positive answer to another. Chapter 3 introduced them and cited the same source; here they are placed against the three rates.

- **Verification** asks: did we build the model we meant to? Does the code implement the six lines, does the mesh represent the drawing, does the state machine have the transitions the operating rules specify. It is a software question, answered by tests, and it is independent of the asset.
- **Calibration** asks: what parameter values make this model match this asset? It is the middle row of Table 6.9, and this section's business.

- **Validation** asks: does the calibrated model match this asset well enough for this decision? It is answered by comparing predictions against readings the calibration did not use, and because a twin's asset keeps changing, it is not a step but a continuous process, performed periodically or on demand, and possibly at more than one level of the system [48].

Validation's evidence – what was compared, how far apart it was, and whether that is close enough – is the credibility chapter's subject and is left to it. This section's claim is narrower: calibration is the activity, verification is its precondition, and validation is its test.

6.9.3 Parameter estimation, or discrepancy

When the residuals grow there are two things a calibration can do, and the Probabilistic Models and Bayesian Updating section met both. **Parameter estimation** assumes the model's form is right and moves its parameters until the residuals shrink: refit k_{th}, update the belief over μ. **Discrepancy** admits the form is wrong by a smooth unknown amount and learns that amount beside the parameters. The choice is a judgement about the residual's shape. A residual that is a constant offset, or that scales with an input the model already has, is a parameter; a residual with a shape no parameter can produce – the oil warming faster at high power than the linear loss rule allows, at every parameter value – is discrepancy, and moving parameters to chase it will make some other prediction worse. Recording which of the two a calibration did is part of the audit below.

6.9.4 Triggers

A slow schedule – refit each season – is the simplest trigger and often the right one for the field model, whose update costs hours. Two others are worth naming.

Drift indicators watch the residual for a change and start an update when one is seen. The twin literature's drift-based approaches compute the error between the model's predictions and recent measurements, track it over a window, and update the deployed model when the error trend crosses a threshold, so that an operational change in the asset is not left unmodelled until the next scheduled refit [27]. For the rotor-and-oil model, a drift indicator is a running mean of the oil-temperature residual; for the bearing, it is the monthly update itself.

Evolution triggers come from the asset's life rather than its data. A repair, a replaced component, a firmware change in the controller, a move to a new lifecycle stage: each says the model *may* need to change before any residual has had time to show it. The twin-engineering literature places evolution at the centre of the discipline and identifies the triggers of a twin's evolution as the events in the actual twin's lifecycle that the twin, as a software system with its own lifecycle, must follow [34]. Some evolution triggers call for a parameter refit; a gearbox swap with a different ratio calls for reconfiguration, and the trigger's job is to raise the question, not to answer it.

6.9.5 What to log

A model update that cannot be audited is a model that cannot be believed, and the credibility chapter will ask for the log. Write down, for every update, at least:

1. **Which model and which version** – of the code, the mesh, the surrogate's training table.
2. **Which parameters moved, from what to what**, and whether a discrepancy term was learned alongside them.
3. **What data the update used** – which readings, over which interval, with which quality flags from Chapter 5 – and what it deliberately excluded.

4. **What triggered it** – the schedule, a drift indicator's value, or an evolution event.
5. **What the residuals were before and after**, on data the update did not use.
6. **Who or what approved it**, if the update was not automatic.

The log is the middle row of Table 6.9 made inspectable, and it is what lets a reviewer six months later tell a model that learned something from a model that was quietly bent to fit a broken sensor.

6.10 Choosing a Model Type

Five model types, four criteria, and a decision to serve. This section is one table and four questions, and it closes with the turbine's answer.

6.10.1 The table

Table 6.10 puts the five types against the four criteria, the expertise each needs and the time each takes to run. Read it as a menu, not a ranking: every row is the right answer to some decision and the wrong answer to others, and the National Academies' report says the same in its own words – a twin's representation should be fit for purpose in model type, fidelity and resolution, chosen and often adapted to the decision and the computational constraints, which may mean high-fidelity, simplified or surrogate models or a mixture [90].

Table 6.10: The five model types against the four criteria, the expertise each needs and its run time. No row passes everything; the twin carries several.

Model type	Answers	Evaluable from readings	Updatable	Time budget	Honest about uncertainty	Expertise needed	Typical run time
lumped first-principles	how much, for a few quantities	yes, from the operational stream	parameters refit, state estimated	passes every pattern	no, unless wrapped	the domain's physics	microseconds a step
discrete-state and discrete-event	which mode; when	yes, from the event log	durations and frequencies re-estimated	passes every pattern	yes, once probabilities are on the arrows	the operation's rules	microseconds an event
high-fidelity field	how much, everywhere	yes, from loads and accelerometers	model updating	fails below certify	yes, once quantified; the most honest of the tower	meshing, a specialist package	minutes to days a run
reduced-order and surrogate	the field model's answer, fast	as the field model	rebuilt when the field model is	passes every pattern	reduced: via the estimator; surrogate: only a Gaussian process, and only in its training region	the field model's, plus projection or fitting	microseconds a step

Model type	Answers	Evaluable from readings	Updatable	Time budget	Honest about uncertainty	Expertise needed	Typical run time
probabilistic, Bayesian	how sure; how long, as a spread	as the model it wraps	it *is* updating	table: instant; sampling: slow schedule	yes, by construction	probability, and the model it wraps	a table, or thousands of evaluations

6.10.2 Four questions

Walk down Table 6.10 with these, in order; each one removes rows.

1. **How much, or which mode?** A question about quantities wants a rate rule or a field; a question about modes, sequences and waiting wants a state machine, an event queue or a Petri net. If both – which mode, and how much in each – it is a hybrid system, and the mode machine picks the rate rules.
2. **How many states does the decision need?** If a handful of lumped quantities can carry it, write the lumped model and stop; it is the cheapest to write, run and keep current. If the decision is about a body – where a stress is, whether a stiffness has changed – it needs a field, and the field model must at least be built.
3. **Does the time budget allow the model's run time?** Take the decision's latency from Chapter 1 and the model's run time from the table, and compare, with the real-time factor of the appendix. If the field model does not fit, reduce it or replace it, and accept that the field model still has to exist first.
4. **Does the decision need a spread, or will a number do?** Monitoring and diagnosis usually take a number and a threshold. Prediction, decision and anything with the word "when" in it needs a spread, and the answer is to wrap whatever model the first three questions chose in the probabilistic type: parameters as beliefs, readings as updates.

The fourth question is the one most often skipped, and skipping it is how a twin comes to report a date for a bearing that fails four months early with no warning the twin's own numbers could have given.

6.10.3 The turbine's answer

Table 6.4 promised the reasons. The power and thermal expectation is a how-much question about two quantities on a ten-minute budget: lumped, and the diagnostic residual falls out of it. Availability and repair planning is a which-mode-and-when question: the mode machine and the vessel's event queue, with weather windows as random events so that the lost-production answer is a spread. Structural integrity is a question about the tower as a body on a seasonal budget: the field model, updated from the accelerometers. Load monitoring every second is the field model's question on a one-second budget: the reduced tower, with the fatigue surrogate for the ten-minute totals. Bearing replacement is a "how long" question with money riding on the confidence: the Wiener model with Bayesian updating of the drift. Five decisions, five model types, one turbine – and the published twins split the same way, the structural twin running a Kalman filter over a linear model of the structure to estimate loads that no sensor measures [164] while the diagnostic twin of the same class of turbine watches its gearbox temperatures for anomalies [22]. A twin is not a model. It is the models a set of decisions needs, kept current at three rates.

6.11 Modelling the Plant Demonstrator

The one section where the rig of Chapter 2 appears. It is here to show that the five model types are not reserved for turbines: the rig's own equations already fall onto the first two, the third and fourth are declined for a stated reason, and the fifth is what the rig's residual has been waiting for. Chapter 2's symbols are used unchanged.

A lumped model, already in sampled time. Chapter 2 wrote the pot as two equations: with no dose the moisture falls at the drying rate, and a dose raises it by the dose response times the millilitres delivered. Read them as this chapter reads a model. The state record is one number, the moisture m. The input record is the dose V the twin asked for. The parameter record is the drying rate d and the dose response g. The step function is the prediction rule Chapter 2 already wrote, from one check to the next T_c hours later:

$$m_{k+1} = m_k + g\,V_k - d\,T_c \tag{6.7}$$

Equation 6.7 is Equation 6.1 with the rig's names in it, and it is already an Euler step: Chapter 2's drying rule "moisture falls by d per hour" is a rate rule, and multiplying d by the interval T_c and subtracting is Euler's step with $h = T_c$. The rig's controller, in other words, was running a lumped first-principles model in sampled time from the first chapter that described it; this chapter has only named the parts. The output function is the identity, $z = m$, because the probe reads the state directly – which is the rig's great simplification and the turbine's great complication.

The controller as a state machine. The watering strategies of Chapter 2 are a state machine with four modes: waiting for the next check, reading the probe, deciding a dose, pumping. The guard on the transition from deciding to pumping is $m < m_t$; its action is to compute V by the dose rule and hold the relay for the pump time the stored calibration gives. The transition from pumping back to waiting is timed, by that pump time. Nothing about this needed a queue, because there is one plant and one pump; the moment the rig has two plants sharing one pump, the shared pump is a resource and the Petri net's tokens are the right picture, which is the exercise this chapter declines to set on the rig and sets on a lift instead.

Why no field model of the soil. Moisture in a pot is a field: wetter at the bottom, drier at the surface, drier again near the roots that drink. A field model of the soil would compute it at every point, and the sensor reads one point. The decision – dose or do not dose, once every six hours, from one probe – cannot use a field's worth of information, and the field model's parameters, the soil's conductivity and the roots' uptake, are not things the rig can measure to calibrate. By the second and third of the four questions the field model is declined: one state carries the decision, and the cost of the model would buy nothing the decision can spend. The published greenhouse twin that is this rig's closest relative makes the same choice, running a simulation model of the greenhouse that calibrates the frequency or duration of watering to how quickly the soil in the pots dries, with an asset model and a time-series database beside it, rather than a model of the soil as a body [53].

Why no surrogate. A surrogate stands in for a model too slow for the decision interval. Equation 6.7 runs in a microsecond on a six-hour interval. There is nothing to replace, and building a surrogate would add the validity-region warning of the Reduced-Order and Surrogate Models section for no gain.

The dose response as an uncertain parameter. Chapter 2's residual, $r = \hat{m} - m_{\text{obs}}$, was left as a number to watch. This chapter says what to do with it. The dose response g is not known for this pot; it depends on the pot's volume, the soil and the season. Give it three candidates, 0.08, 0.10 and 0.12 percentage points per millilitre, equally likely to begin with. Deliver an 80-millilitre dose and predict the rise under each: 6.4, 8.0 and 9.6 points. Observe a rise of 8.9 points, with the check-to-check scatter, drying included, taken as a Gaussian of standard deviation 1.0. The

likelihoods, from the Gaussian density at 8.9, are in the ratio 0.044 to 0.667 to 0.783, and with a flat prior the posterior is 0.03, 0.45 and 0.52: two doses in, the twin already believes g is 0.10 or above and has nearly ruled out 0.08. That posterior is what Chapter 3's Stage 2 – the polling fault detector whose residual a technician acts on – should feed: not a threshold on one residual, but a belief over g that a run of residuals sharpens, and that says, when the belief drifts downward over weeks, that the pot is responding less to each millilitre than it did, which is either roots filling the pot or a pump delivering less than it claims. Which of the two it is, the pump-time model of Chapter 2 and the two candidates for where the residual appears – between doses or at them – can tell apart; the belief is what tells the technician to look.

6.12 Faded Examples

Five examples, one per model type, each on the second system chosen for that type, and each with less of the work done for you than the one before: the first is worked in full, the next two leave one step to you, and the last two are posed as problems. None of them is the rig.

6.12.1 Example 1, worked in full: a battery cell as an equivalent circuit

Lumped first-principles. A lithium-ion cell in a bus, whose twin must report how much charge is left and how healthy the cell is. The cheapest model with the physics in it is the **equivalent circuit**: the cell as a voltage source whose voltage depends on the state of charge, in series with a resistor for the instantaneous losses, in series with a resistor-capacitor pair for the slower polarisation that makes the terminal voltage sag over tens of seconds under load. Model-based battery prognostics work on exactly such equivalent-circuit models of the cell voltage, or on physics-based degradation models, and both generalise across cells of one chemistry because they combine known physics with voltage, current and resistance data [177]. Table 6.11 names the parts.

Table 6.11: The battery cell's equivalent-circuit model. The two parameters marked "and health" are the ones a battery twin's digital state carries.

Symbol	Meaning	Unit	Role	Value used
SoC	state of charge, fraction of capacity remaining	–	state	0.800
V_1	voltage across the polarisation pair	volts	state	0
I	current drawn, positive on discharge	amperes	input	3.0
Q	capacity	coulombs	parameter, and health	10800 (three ampere-hours)
R_0	series resistance	ohms	parameter, and health	0.020

Symbol	Meaning	Unit	Role	Value used
R_1, C_1	polarisation resistance and capacitance	ohms, farads	parameters	0.015, 2000
OCV(SoC)	open-circuit voltage at this state of charge	volts	a lookup table, part of the parameters	3.95 at 0.800
V_t	terminal voltage, what the pack's sensor reads	volts	output	–
h	step	seconds	the loop's	1

Three rules from conservation. Charge out of the cell is charge no longer stored, so the state of charge falls at the current over the capacity:

$$\frac{d\,SoC}{dt} = -\frac{I}{Q}$$

The polarisation capacitor stores charge that the current pushes in and its resistor lets leak away, so its voltage rises with the current and decays with its own time constant:

$$\frac{dV_1}{dt} = \frac{I}{C_1} - \frac{V_1}{R_1 C_1}$$

And the output function is the circuit read from its terminals: the open-circuit voltage less the two drops,

$$V_t = \text{OCV}(SoC) - R_0\, I - V_1 \tag{6.8}$$

One step. With the values of Table 6.11, the state of charge falls by $3.0/10800 = 0.000278$ in one second, to 0.79972; the polarisation voltage rises by $3.0/2000 = 0.0015$ volts from zero, its decay term being zero at the start; and the terminal voltage by Equation 6.8 is $3.95 - 0.020 \times 3.0 - 0.0015 = 3.8885$ volts. Take another step and the decay term is no longer zero: V_1 rises by $0.0015 - 0.0015/30$, a little less, and will level off at $I R_1 = 0.045$ volts after a few time constants of thirty seconds. That sag, from 3.89 to about 3.845 volts under a steady three amperes, is what the pack's voltage sensor will show, and the model says why.

Why the parameters are the twin's digital state. Nothing in the step moves Q or R_0; they are parameters. But over months the cell fades and Q falls, and it ages and R_0 rises, and the twin's whole purpose is to know by how much. The twin literature names exactly these – the capacity and the internal resistance as state-of-health parameters of a lithium-ion cell – as the digital state a battery twin estimates through noisy voltage and current readings [27]. In this chapter's terms: the parameters of the lumped model are the beliefs of the probabilistic one, and the Keeping a Model Current section's middle row is where the battery's value lives.

6.12.2 Example 2, one step left to you: a 3D printer's modes

Discrete-state and discrete-event. A desktop 3D printer in a shared workshop, whose twin must report what the printer is doing and how long the queue of jobs will take. Its modes have been published as a Unified Modeling Language (UML) state machine: the printer starts in **StandBy**, becomes **Connected** when a device connects, becomes **Operational** on receiving a print file

and preheats before printing, can be paused and resumed or cancelled while printing, returns to StandBy when the job finishes, can enter **Error** from any state, and is reset from Error back to StandBy [178]. That is Figure 6.2 with different words on the boxes, and drawing it is the worked half: six modes, transitions with guards ("file received", "preheat temperature reached", "job finished", "error raised", "reset").

The step left to you. Extend it in two directions. First, add the queue: jobs arrive at random times with a print duration each, and the printer serves them one at a time. Write the events – job submitted, print started, print finished, error raised, reset – as timestamped entries in a priority queue in the manner of Table 6.7, and say what the model's state is when the queue holds three jobs and the printer is in Error. Second, make it hybrid: inside the printing mode, the extruder's temperature obeys a rate rule of the gearbox-oil shape – heater power in, loss to the air out – and the guard "preheat temperature reached" fires when that rate rule's state crosses a threshold. The manufacturing-twin review records a generic hybrid automaton of the continuous and discrete-event dynamics of additive processes as precisely this construction [91]. Write the rate rule, name the mode in which it runs, and say which transition its state fires.

6.12.3 Example 3, one step left to you: an industrial oven with a conveyor

High-fidelity field. Steel bars ride a conveyor through a furnace and must leave at a target temperature. A twin of such a furnace has been published for testing: the furnace is divided into five zones of induction heaters, four coils each, with a pyrometer between coils reading the bar temperature, and when the line downstream stalls the furnace drops into a holding mode with reduced coil power and the bars oscillating slowly to hold their temperature [179]. The engineering-design literature describes the model such a furnace's remaining-life twin needs: a computational fluid dynamics model of the temperature distribution in the furnace chamber, a finite-element model of the stress in its weld seams, qualified reduced-order versions of both, and real temperature sensors near the hot parts feeding virtual sensors inside them [62].

Worked half. The field is the temperature at every point of the chamber's gas and the bars' steel; the mesh covers both; the inputs are the coil powers and the belt speed; the parameters are the steel's conductivity and heat capacity and the gas's flow properties; the output function reads the temperature at the pyrometers' spots.

The step left to you. Two judgements a meshing engineer makes, made here in words. First, where must the mesh be fine? Consider where the temperature changes fastest in space – at a bar's surface as it enters a zone, around each coil – and where the decision needs the value – at the pyrometer spots and at the weld seams – and say where you would spend the nodes and where a coarse mesh will do. Second, what does the belt speed do to the field? A faster belt gives each bar less time in each zone, so the same coil power produces a lower exit temperature and a steeper gradient from a bar's surface to its core; say what the holding mode, with its slow oscillation, does to that gradient, and why the twin's operators might prefer it to stopping the belt.

6.12.4 Example 4, posed as a problem: reduced models and a surrogate for capacity fade

Reduced-order and surrogate. Return to the battery. Over hundreds of charge-discharge cycles its capacity Q fades. The battery prognostics literature's earliest models of that fade are **algebraic reduced-order models**: a formula for capacity against cycle number with two or three fitted parameters, assumed to follow an algebraic trajectory such as a power law, which, paired with sampling over its parameters, captures model-form and parameter uncertainty in a way a network ensemble does not [163]. A recent framework goes one step further and fits a machine-learning

model that predicts the fade model's parameters from **early-life features** – a few numbers computed from the first hundred cycles – so that a cell's whole capacity trajectory is predicted before most of it has happened, with under two per cent error on cells like the training set and under four on cells unlike it [180].

The problem. (a) Write a two-parameter algebraic fade model, $Q(n) = Q_0\,(1 - a\,n^b)$, with n the cycle number, and say which of this chapter's model types it is and what it reduced – what is the "field model" it stands in for, and what did it keep. (b) The machine-learning model from early-life features to a and b is a surrogate. Say what computational model it mimics, and then apply both warnings of the Reduced-Order and Surrogate Models section to it: what is it *not*, and where is it valid. (c) State what the twin must still hold and update for the pair to count as part of a twin rather than as a calculator.

6.12.5 Example 5, posed as a problem: capacity fade as a Wiener process

Probabilistic, Bayesian updating. The same fade, modelled the way the Probabilistic Models and Bayesian Updating section modelled bearing wear. Capacity falls on average by a fade rate μ per cycle and wanders by σ per square-root cycle; the cell is retired when Q falls below eighty per cent of its rated value.

The problem. (a) Write the step, in the form of Equation 6.6 with the battery's names. (b) Give μ three candidate values from a fleet – say 0.010, 0.015 and 0.025 per cent of rated capacity per cycle – and a prior, observe a fade of 0.8 per cent over the last fifty cycles with a scatter you choose, and fill in a table of the shape of Table 6.8. (c) From the posterior, describe the remaining-life distribution in cycles for a cell now at ninety per cent: three centres, their weights, and what σ does to each. (d) Say in one sentence why the answer to (c) is more useful to the bus operator than the single number Example 4's fade model returns, and in a second sentence what Example 4's model gives that this one does not.

6.13 Summary

Six items, returning to the six objectives.

1. **Every model is four records and two functions** (objective 1). A state – the smallest record that, with the future inputs, fixes the future – an input, a parameter set and an output, a step function from state, input and interval to next state, and an output function that reads what a sensor would see. Time enters the step three ways, continuous, sampled and event-driven; a step returns a value or a distribution; and a model is white, grey or black by how much physics is written in it. Four criteria judge a model inside a twin: evaluable from this asset's readings, updatable, inside the decision's time budget, honest about uncertainty.
2. **Five model types, chosen per decision** (objective 2). Lumped first-principles for how much about a few quantities; discrete-state and discrete-event for which mode and when; high-fidelity field for how much everywhere; reduced-order and surrogate for the field model's answer inside the time budget; probabilistic with Bayesian updating for how sure. The turbine needed all five, one per decision, and the choosing table's four questions walk from a decision to a row.
3. **A rate rule becomes a loop by Euler's step, and the step size is set by the fastest state** (objective 3). The rotor's speed and the gearbox oil's temperature share one step, so the oil is stepped six hundred times per reading to move a hundredth of a degree; that is what a small step costs, and the appendix's better solvers are the remedy. One Euler step of

a linear lumped model is the discrete state-space form, handed to the state and estimator chapter.

4. **A mesh turns a body into a million states; a mode is a shape the body prefers to move in, now computed rather than assumed; a reduced basis is the few modes that carry the motion** (objective 4). The field model is worth paying for when the decision is about the body – a changed stiffness, a stress at a weld – and the reduced model or the surrogate is how its answer fits a one-second budget, with the two warnings: a surrogate mimics a model, not the asset, and is valid only where it was trained.
5. **Degradation is a random walk with drift, remaining life is a distribution, and a reading updates a parameter by Bayes' rule** (objective 5). Prior times likelihood, normalised: one month's vibration rise moved the bearing's drift from probably slow to probably normal, and the remaining life became a weighted mixture of three centres rather than a date. Model-form discrepancy is the admission that the form is wrong by a smooth unknown amount, learned beside the parameters; the probabilistic graphical model is the whole twin drawn as a chain of such updates over a digital state.
6. **Three rates, three homes** (objective 6). State changes every reading and belongs to the estimator in the state and estimator chapter; parameters change on a slow schedule or a trigger and belong to this chapter, under calibration, with drift indicators and evolution triggers to start an update and a six-line log to audit it; structure changes when the asset changes and belongs to configuration. Verification, calibration and validation are three questions, and the credibility chapter takes the third.

The sentence to carry forward: **a model gives the twin its expectation; the estimator, next, is how the twin corrects that expectation with what the sensors say.**

6.14 Exercises

Solutions or hints follow the exercises; each says which. The objective each exercise serves and its difficulty are given in brackets. None of the exercises uses the plant demonstrator.

Exercise 1 (Objective 1, easy). A water tower is filled by a pump and drained by a town's demand. Name its state, inputs, parameters and output, and write its step function for an interval h in the style of the turbine's `step` function. Say which quantity you were tempted to call state and is not, and why.

Exercise 2 (Objective 1, medium). A lift in a ten-storey building serves calls from the floors. Name the state record of a model that answers "how long will I wait", and say which of the three notions of time its step should use and why. Then name the state record of a model that answers "how hot is the motor", and say why the two records share nothing.

Exercise 3 (Objective 3, easy). The gearbox oil's rate rule has a time constant $C_{\text{th}}/k_{\text{th}} = 2000$ seconds. Step it with $h = 1$, $h = 600$ and $h = 3000$ seconds from $T_{\text{oil}} = 48$ towards its steady value of 52 degrees, for one step each, and say what goes wrong at the largest step. Which appendix section explains it?

Exercise 4 (Objective 3, medium). Rewrite the turbine's two-state step as the matrix form Equation 6.5 with $h = 600$ seconds, compute F, and apply it once to $x = \begin{bmatrix} 1.70 \\ 48.0 \end{bmatrix}$ with $u = \begin{bmatrix} 6.0 \times 10^4 \\ 4.0 \times 10^4 + 1.0 \times 10^3 \times 12 \end{bmatrix}$. Compare the oil temperature you get with six hundred one-second steps of the worked example, and say which is closer to the truth and why.

Exercise 5 (Objective 2, medium). For each decision below, choose a model type and defend it against the four criteria in one sentence each: (a) a data centre's cooling plant must decide every minute whether to start a second chiller; (b) a bridge owner must decide, once a

year, whether a deck can carry a new heavier vehicle class; (c) a hospital must estimate how many beds will be free at 08:00 tomorrow.

Exercise 6 (Objective 2, hard). Draw the Petri net for one lift serving two floors whose doors cannot both be open while the car is moving. Name the places, the transitions and the tokens, and say what the marking is when the car is at floor 1 with its door open and a call is pending at floor 2. Then say, in one sentence, why a finite state machine of the same system needs more modes than the Petri net has places.

Exercise 7 (Objective 4, easy). Explain to a colleague, in three sentences and no equations, what a mesh, a mode and a reduced basis are, using the tower of Figure 6.3 as the instance.

Exercise 8 (Objective 4, medium). A twin's fatigue surrogate was fitted on runs with wind speeds from 4 to 25 metres per second and wave heights up to 6 metres. A storm arrives with 32 metres per second and 8-metre waves. Say what the surrogate will return, what a Gaussian process surrogate would return that a polynomial would not, and what the twin should do for the duration of the storm.

Exercise 9 (Objective 5, medium). A pump's seal leaks at a rate modelled by Equation 6.6 with drift μ in millilitres per hour per week and a replacement threshold of 60 millilitres per hour. The current leak is 24 millilitres per hour. Fleet history gives μ three candidates, 3, 6 and 12, with prior 0.4, 0.4 and 0.2. Last week the leak rose by 7 millilitres per hour; take the weekly scatter as a Gaussian of standard deviation 3. Fill in a table of the shape of Table 6.8 and describe the remaining-life distribution in weeks.

Exercise 10 (Objective 5, hard). Repeat Exercise 9 for a second week in which the leak rose by 2 millilitres per hour, starting from the posterior you found. Then say, in two sentences, why the two-week result is not the same as a single update with a two-week rise of 9, and what that says about the value of frequent readings.

Exercise 11 (Objective 6, easy). Sort the following into the three rows of Table 6.9, naming the owning chapter for each: the rotor speed; the gear ratio after a gearbox swap; the heat-loss coefficient after a cooler is cleaned; the belief over the turbine's mode; a new accelerometer added at mid-height; the bearing's drift.

Exercise 12 (Objective 6, medium). A colleague proposes that the state estimator be allowed to adjust the rotor inertia J at every reading "so the model always fits". Using the three rates and the verification-calibration-validation distinction, say what will go wrong, which trigger and schedule you would propose instead, and which of the six log entries would let a reviewer catch the problem after the fact.

6.14.1 Solutions and hints

Exercise 1 (solution). State: the water level, or equivalently the volume stored. Inputs: the pump's flow when on, and the town's demand. Parameters: the tank's cross-section, which turns a volume into a level, and the pump's rated flow. Output: the level the tank's float sensor reports. Step: level $+ h\,(\text{pump flow} - \text{demand})/\text{area}$. The temptation is to call the pump's on-or-off status state; it is an input, decided outside the tank, unless the model includes the controller, in which case it becomes a mode in a hybrid model.

Exercise 2 (hint). For waiting time the state is the car's floor and direction, its door state, and the set of pending calls – discrete things – so the step is event-driven, popping calls and arrivals from a queue. For the motor's heat the state is a temperature and the step is a rate rule in continuous or sampled time. They share nothing because the decisions share nothing; a twin of the lift holds both.

Exercise 3 (solution). The rate is $(52 - 48)/2000 = 0.002$ degrees per second. With $h = 1$ the oil reaches 48.002; with $h = 600$, 49.2, still short of 52 and plausible; with $h = 3000$ it reaches 54, *overshooting* the steady value it can never physically exceed, and the next step would swing

back below it. A step longer than the time constant makes Euler's step unstable; the appendix's *From an Equation to a Loop*, under *Worked example 2* and *Better Solvers*, shows it and the implicit step that cures it.

Exercise 4 (hint). With $h = 600$ seconds the two tables are:

$$F = \begin{bmatrix} 1 & 0 \\ 0 & 0.7 \end{bmatrix}, \qquad G = \begin{bmatrix} 1.0 \times 10^{-4} & 0 \\ 0 & 3.0 \times 10^{-4} \end{bmatrix}$$

The second diagonal entry of F is $1 - 600 \times 1000/(2.0 \times 10^6) = 0.7$. One step gives $\omega_r = 1.70 + 600 \times 0.010 = 7.70$ – which is absurd, because the torque imbalance would not stay constant for ten minutes; the controller would act – and $T_{oil} = 0.7 \times 48 + 3.0 \times 10^{-4} \times 5.2 \times 10^4 = 49.2$. Six hundred small steps give about 49.1, and the smaller-step answer is closer because it follows the decaying rate rather than holding the initial rate for the whole interval. The rotor row is the lesson: a step must be short compared with how fast the *inputs* change, too.

Exercise 5 (hint). (a) A lumped thermal model of the plant, possibly wrapped in a mode machine for chiller states: evaluable from the plant's sensors, refit as the plant fouls, microseconds against a one-minute budget; honest about uncertainty only if wrapped. (b) A field model of the deck, updated from accelerometers on a yearly schedule: the decision is about the body, the budget is a year, and the field model's quantified uncertainty is what a certification needs. (c) A discrete-event model of admissions, discharges and transfers with durations from the log, run many times for a spread: which-mode-and- when, cheap, and honest once the durations are distributions.

Exercise 6 (hint). Places: car at 1, car at 2, car moving, door 1 open, door 1 closed, door 2 open, door 2 closed, call pending at 1, call pending at 2. Transitions: open door 1 (needs car at 1 and door 1 closed), close door 1, depart 1 for 2 (needs car at 1, door 1 closed and door 2 closed), arrive at 2, and their mirrors; serve call 2 consumes the pending-call token when the car arrives with the door open. The marking asked for has one token in car at 1, door 1 open, door 2 closed and call pending at 2. A state machine needs one mode per reachable *combination* of car position, both doors and both calls, which multiplies where the Petri net adds.

Exercise 7 (hint). A mesh cuts the tower into small pieces and keeps a number at each corner, so that the tower's motion is described by many numbers instead of one. A mode is a way the tower likes to sway or twist, with its own tempo, and a tall tower has a few slow ones that carry almost all its motion. A reduced basis keeps only those few, so the twin tracks a handful of tempos instead of a million corners and can still rebuild the whole picture from them.

Exercise 8 (solution). A polynomial surrogate will return a number extrapolated from the edge of its fitted region, with no sign that it is outside it; the number is untrustworthy in either direction. A Gaussian process will return a number with a spread that has grown large, signalling that it does not know. Either way the twin should flag the inputs as out of the validity region, stop adding the surrogate's fatigue increments to the running total for the storm, and fall back to the reduced model driven by the accelerometers, or to a field-model run afterwards on the recorded loads, to fill in the storm's damage.

Exercise 9 (solution). Likelihoods of a rise of 7 with standard deviation 3: for $\mu = 3$, the reading is 1.33 deviations away, density about 0.055; for 6, 0.33 away, about 0.126; for 12, 1.67 away, about 0.033. Prior times likelihood: 0.022, 0.050, 0.007; sum 0.079; posterior 0.28, 0.64, 0.08. Distance to the threshold is 36 millilitres per hour, so the centres are 12, 6 and 3 weeks with weights 0.28, 0.64 and 0.08: most likely about six weeks, a real chance of twelve, a small chance of three, each centre widened by the scatter.

Exercise 10 (hint). Start from the posterior 0.28, 0.64, 0.08 as the new prior; a rise of 2 favours the slow candidate. The two-week result differs from one update on a rise of 9 because the two-week scatter is not 3 but $3\sqrt{2}$, and because two readings constrain the *path* – a fast week

then a slow one is evidence against a steady fast drift that a single two-week total cannot see. Frequent readings buy shape, not just total.

Exercise 11 (solution). State, the estimator's chapter: the rotor speed; the belief over the mode. Parameters, this chapter: the heat-loss coefficient after cleaning; the bearing's drift. Structure, the configuration and manager chapter: the gear ratio after a gearbox swap, because the constraint changes and so may the model; the new accelerometer, because the output function gains a row.

Exercise 12 (hint). An inertia that moves every reading will absorb sensor noise, a stuck speed counter and controller transients into a physical constant that cannot change on that time scale, so the model "fits" and predicts nothing; the state should be estimated every reading and J refit on a schedule, or on a drift indicator over the speed residual, with the residual checked on held-out data before the new value is accepted. Log entries 2 and 5 – which parameters moved, and the residuals before and after on data the update did not use – are what a reviewer would use to see that J had been chasing noise.

6.15 Further Reading

Grouped by model type, saying where a worked example is the book's own construction rather than a source's.

- *What every model has:* [6] for the white, grey and black box sorting as the twin literature uses it; [40] for reading the twin itself as the grey position between model-based and vibration-based monitoring. The four-records-two-functions view, the three notions of time and the four criteria are the book's own framing.
- *The turbine:* [164] for the physics-based twin of a floating turbine validated on a full-scale prototype, whose estimator and virtual sensing the state and estimator chapter will return to; [22] for the diagnostic twin of an operational floating turbine and the SCADA signals it reads; [165] for the offshore-structures programme of predictive modelling; [162] for what SCADA data looks like at farm scale. The six-line rotor-and-oil model, its numbers and the vessel-dispatch table are the book's own.
- *Lumped first-principles:* [166] for lumped-parameter models, the bond-graph analogy across domains and the acausal style, at book length; [167] for Modelica models reaching a twin as compiled units in a DevOps pipeline; [168] for a whole system decomposed into lumped component models joined by constraints; [169] for an equation-based building model coupled to a sequential simulator and what each side gives up.
- *Discrete-state and discrete-event:* [166] again, for DEVS and Petri nets with worked code; [12] for choosing a formalism as a design stage; [91] for hidden Markov models and hybrid automata of machine states in manufacturing twins; [49] and [170] for discrete-event twins of production and of a hospital.
- *High-fidelity field:* [165] and [40] for finite-element model updating of offshore structures from measured modal parameters; [171] for the aircraft wing calibrated on its first two bending modes; [37] for building twins generated from the building information model; [172] for model updating from operational modal analysis on a twin platform; [91] for the finding that fidelity's computational price often makes such models unsuitable inside a twin.
- *Reduced-order and surrogate:* [92] for what a surrogate is and the argument that a surrogate alone is not a twin; [20] for reduced models with a state estimator as virtual sensors, including the turbine blade; [66] for a library of component reduced models; [173] for interpretable, reusable reduced models in manufacturing; [174] for a multi-fidelity surrogate

inside Bayesian updating; [90] for fit-for-purpose fidelity and the open problem of adapting surrogates under time constraints. The fatigue surrogate is the book's own construction.

- *Probabilistic and Bayesian:* [27] for the digital state and the stochastic-process degradation models; [11] for Bayesian updating, the Kennedy and O'Hagan discrepancy framework and the sampling methods named here and taught nowhere in this book; [26] for remaining useful life in predictive-maintenance twins; [171] and [175] for the probabilistic graphical model of a twin, the figure this chapter redrew in its own terms. The three-candidate table and the bearing's numbers are the book's own.
- *Keeping a model current:* [54] for architectural self-adaptation when the asset changes stage; [176] for evolving a twin's schema and data together; [48] for verification and validation as a continuous process; [27] for drift-based updating; [34] for evolution triggers. The three-rates table and the six-line log are the book's own.
- *The faded examples:* [177] and [27] for the battery's equivalent circuit and its health parameters as digital state; [178] for the 3D printer's published state machine; [62] and [179] for the furnace's field model and its zones and coils; [163] and [180] for algebraic capacity-fade models and the early-life surrogate. The battery's circuit values and the worked step are the book's own.
- *The rig:* [53] for the published greenhouse twin whose watering calibration is the rig's closest relative.

Chapter 7

Simulation Techniques and Simulators

7.1 Before You Start

What you are assumed to know. You write software for a living, you read a function signature and know what its arguments and return value commit it to, and you remember basic probability from a first course. You have read the previous six chapters. From Chapter 3 you have three words: a **model** is a description of how something behaves, and it does not run; a **simulation** is one execution of a model over time from a starting state; a **simulator** is the software that performs simulations. From Chapter 4 you have the seven blocks, in particular the **simulators** block – the block that executes models – and the **models** block, which that chapter reduced to one interface: given a state and a horizon, return a predicted state. From Chapter 5 you have the tuple, the sampling interval T_s, the event time, the hot, warm and cold paths, and the age of twin. From Chapter 6 you have the five **model types** – lumped first-principles, discrete-state and discrete-event, high-fidelity field, reduced-order and surrogate, and probabilistic with Bayesian updating – each with its step function, the four criteria a model is judged on, and the choosing table. The offshore wind turbine of that chapter is this chapter's running example too, with the same numbers. From Chapter 2 you have the plant rig, which reappears once, in the one section where it is allowed.

What you are not assumed to know. Numerical analysis, queueing theory, statistics beyond a first course, and any simulation tool. Every mathematical result this chapter leans on is recalled in one clause and pointed at the section of the mathematics appendix that builds it: *From an Equation to a Loop* for what a solver is and what Euler's step does; *Better Solvers* for Heun's step, the implicit step and adaptive stepping; *Stiff Equations for Digital Twins* for why a step size is forced; *Many Quantities at Once* for the state-space form, dimension and modes; *Probability and Distributions* and *State Machines with Probabilities* for the Gaussian, Bayes' rule and the belief that leaks away as a chain is predicted forward; and *What a Solver and a Simulator Do* for the event queue, the cost of a run and the real-time factor. If a paragraph below feels like it is skipping a step, the appendix section named beside it is where the step is.

What this chapter covers. Chapter 6 said what goes in Chapter 4's model slot. This chapter says what executes it. It first takes a simulator apart into four pieces – an engine, a step, a clock and an input-output contract – and gives four criteria a simulator is judged on, parallel to the four that chapter gave a model. It then walks the five model types again, in the same order, as five **execution problems**: running rate rules, running an event queue, running both at once, running one model many times, and running a field model that cannot be run at all inside a twin – followed by the section on how the field model's answer is bought back cheaply. Three sections then make the chapter usable: time, where the real-time factor stops being a definition

and becomes a placement decision; the concrete interface a twin calls a simulator through, with five run modes; and a choosing table with four questions and the numerical checks that make a run trustworthy. The rig, three faded examples, a summary, exercises and further reading close the chapter.

What this chapter deliberately does not cover.

- **Co-simulation, its standard interfaces, and distributed or federated simulation.** Running two or more simulators together, exchanging values at agreed instants, is a large and well-standardised subject, and it belongs with the standards it rests on: the Functional Mock-up Interface and the High Level Architecture live in the platforms and standards chapter of Part III, and co-simulation is taught there. This chapter is about one simulator, and says nothing further about joining two.
- **The mathematics of any solver.** Recalled in one clause each, with a pointer to the appendix section that builds it.
- **State estimation, the Kalman filter and the correction step.** The state and estimator chapter's. This chapter produces the prediction that an estimator corrects, and stops at the handover.
- **Whether a run deserves to be believed.** The credibility chapter's. This chapter covers numerical verification only – did the arithmetic converge, is the answer the model's and not the step size's – and hands validation on in one sentence.
- **Model updating and parameter re-estimation.** Chapter 6 kept them.
- **Neural, hybrid physics-plus-network and generative surrogates.** Parked for the AI chapter; named here only as parked.
- **Any simulation tool's syntax.** Named where a published system used one, taught nowhere.

Learning objectives. By the end of this chapter you will be able to:

1. **Name** the parts of a simulator and **state** what a twin asks of one.
2. **Choose** a solver and a step size for a rate-rule model and **defend** the choice against stiffness and a time budget.
3. **Decide** whether a model needs a solver, a scheduler or both, and **say** what a mode switch costs.
4. **Design** an ensemble run – how many replications, which seeds, and what the output is.
5. **Compute** a run's real-time factor and **place** the simulator as embedded, edge or batch.
6. **State** what a run's provenance must record and **name** the numerical checks that make a run trustworthy without claiming it valid.

Every section that follows serves at least one of those six, and the exercises at the end say which.

7.2 What a Simulator Is, Architecturally

Chapter 4 gave the models block one interface – give me a state and a horizon, return a predicted state – and said nothing about what sits behind it. Behind it is a simulator, and a simulator is four things. Figure 7.1 draws them.

The engine owns the loop. It holds the current simulated time and the current state, decides when to stop, decides when to record, and calls the step. Nothing else in the simulator knows what time it is.

The step advances the model once. It comes in exactly two kinds, and the whole of this chapter is a consequence of that fact. A **solver** takes a state, asks the model for its rate, and

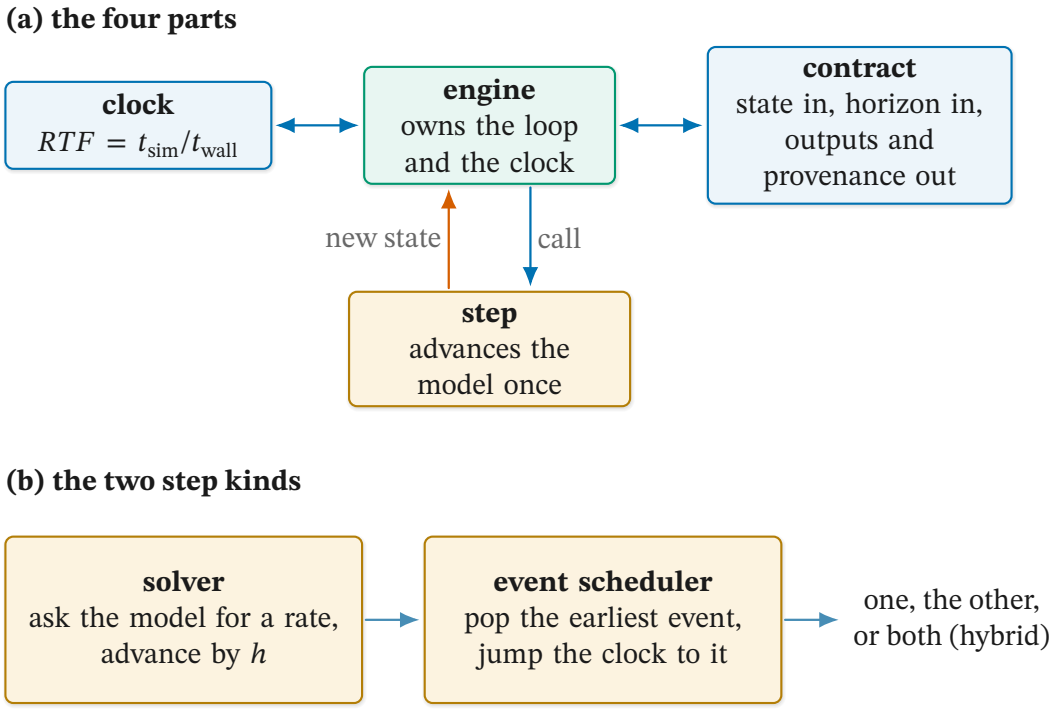


Figure 7.1: The four parts of a simulator, and the two interchangeable step kinds. Panel (a): the engine owns the loop and the clock, calls the step, and writes outputs through the contract. Panel (b): the step is either a solver, which asks the model for a rate and advances by an interval, or an event scheduler, which pops the earliest pending event and advances the clock to it.

returns the state an interval h later; the appendix's *From an Equation to a Loop* is a solver with h chosen once, up front. An **event scheduler** takes a priority queue of pending events, pops the earliest, advances the clock to its timestamp, runs its handler, and pushes whatever the handler schedules; the appendix's *What a Solver and a Simulator Do* writes the loop out in seven lines. A simulator has one, the other, or – the Hybrid Simulation section below – both, with one driving the other.

The clock decides what one second of simulated time costs in wall-clock time. It is the part a reader new to simulation does not expect to exist, because in an ordinary program there is only one clock. In a simulator there are two, they run at different rates, and the ratio between them is the **real-time factor** *RTF* the appendix defines: simulated time covered per unit of wall-clock time. The Time section below is entirely about that ratio.

The input-output contract is Chapter 4's interface, written out: how the initial state gets in, where the inputs over the horizon come from, what the outputs look like on the way out, and what the run records about itself. The Driving a Simulator from a Twin section below gives it as a function signature.

7.2.1 The four criteria

Chapter 6 judged a model on four criteria. A simulator is judged on four of its own, and they are about execution rather than about representation.

1. **It finishes inside the decision's time budget.** Chapter 1 gave every value pattern a latency; a simulator that answers after the decision was due has not answered. This is the *RTF* question, and it is the one criterion that can be computed in advance rather than argued about.

2. **It is restartable from a state the twin supplies.** A simulator written for design runs starts from a designed initial condition and runs for hours. A simulator inside a twin is handed the state block's current belief every cycle and asked for the next few minutes. If setting the state costs more than the run, the simulator is the wrong one, however fast its inner loop is.
3. **It is deterministic enough to be compared with itself.** Two runs from the same state with the same inputs and the same seed must give the same answer, or the difference between yesterday's prediction and today's says nothing about the asset. This is weaker than bit-reproducibility and stronger than "close enough": what it rules out is a simulator whose answer depends on thread scheduling, on an unseeded random source, or on wall-clock time leaking into the model.
4. **It says what it ran.** The model and its version, the solver and its settings, the initial state, the inputs, the seed, and what the run cost. A number a twin cannot reproduce is a number a twin cannot defend, and the Driving a Simulator from a Twin section makes this a record the run returns rather than a logging convention.

Criteria 3 and 4 are software-engineering criteria, not numerical ones, and they are the two most often missing from a simulator that was written for a design office and then pointed at an asset.

7.2.2 The simulator is usually something you are handed

Chapter 4 observed that a digital twin framework names the blocks it implements and implements the loop rather than the solver. That is worth restating here, because it decides what a software engineer's job on this chapter actually is. You will rarely write a solver. You will choose one, configure it, work out whether it fits the budget, wire the twin's state into it, and be responsible for the answer. The executable-twin literature names the artefact this produces: a self-contained executable digital behaviour of an asset, to be used by anyone at any point in the lifecycle, which requires the model to be fast enough – often real-time – and in most cases requires the model order reduction the Making an Expensive Simulator Cheap section below describes, with the packaged model carrying its execution engine with it and an explicit link back to the parent twin that keeps the two consistent [20]. A simulator, in other words, is increasingly shipped rather than written; the four criteria above are what you check it against on arrival.

7.3 Continuous-Time Simulation: Running the Rate Rules

The execution problem of Chapter 6's **lumped first-principles** model type. The model is a handful of rate rules; the simulator is a solver; the only real decision is the step, and it is worth more thought than it usually gets.

7.3.1 Four steps, priced

The appendix's *Better Solvers* builds these; this section prices them. The price is in **rate evaluations per step**, because for a small model almost all the work is evaluating the right-hand side.

Table 7.1: The four step kinds a twin's solver will offer, priced in rate evaluations. The error column is how the error at the end of a run shrinks when the step is halved; the stability column is what forces a short step whether or not the accuracy needs one.

Step	What it does	Rate evaluations per step	Error falls as	Stable for
explicit Euler	uses the rate at the start of the interval	1	h	h below about twice the fastest time constant
Heun	averages the rate at both ends, guessing the end with an Euler step	2	h^2	roughly the same bound
classical Runge-Kutta	samples the rate four times across the interval	4	h^4	a somewhat larger bound
implicit Euler	uses the rate at the end, so each step solves a small equation	1, plus an equation solve	h	any h
adaptive	estimates its own error and changes h as it goes	varies	to a tolerance you set	it chooses

Read Table 7.1 as an engineering menu. Going from Euler to Runge-Kutta costs four times the work per step and buys a step that can be roughly ten times longer for the same accuracy – a net win, and the reason nobody uses fixed-step Euler in production. Going from explicit to implicit costs an equation solve per step and buys the removal of the stability column, which is the only thing that helps when the model is stiff. Going adaptive replaces "choose h" with "choose a tolerance", which is a better question to ask a domain expert; solvers of this kind take long steps where the dynamics are slow and short ones where they are fast or where the instant something happens has to be located properly, and they are the family customarily used on stiff systems [169]. What a tolerance costs is measurable: one study sweeping a general-purpose solver's relative tolerance from 10^{-2} down to 10^{-12} on the same model reports the step count rather than the elapsed time, because on a small model the elapsed time is buried in operating-system noise – and notes that on a larger model the same effect does appear in the time [181]. That is the honest way to measure a solver setting, and it is worth copying.

7.3.2 Stiffness, on the turbine's own numbers

Chapter 6 wrote the turbine's rotor and gearbox oil as two rate rules:

$$J \frac{d\omega_{\mathrm{r}}}{dt} = Q_{\mathrm{a}} - Q_{\mathrm{g}} \tag{7.1}$$

$$C_{\mathrm{th}} \frac{dT_{\mathrm{oil}}}{dt} = P_{\mathrm{loss}} - k_{\mathrm{th}} \left(T_{\mathrm{oil}} - T_{\mathrm{amb}} \right) \tag{7.2}$$

with $J = 6.0 \times 10^6$ kilogram square metres, $C_{\text{th}} = 2.0 \times 10^6$ joules per kelvin, $k_{\text{th}} = 1.0 \times 10^3$ watts per kelvin and $T_{\text{amb}} = 12$ degrees. Equation 7.1 has no damping term in it at all as Chapter 6 wrote it, and Equation 7.2 has one; that asymmetry is where this subsection starts. The oil's time constant is $C_{\text{th}}/k_{\text{th}} = 2000$ seconds: left alone it closes a third of its gap to the steady value every half hour. That is the slow process.

Chapter 6 took the generator torque Q_{g} as an input, which is right for a model. A *running* turbine sets it: below rated wind the controller holds the tip-speed ratio by commanding torque proportional to the square of the rotor speed, $Q_{\text{g}} = K\omega_{\text{r}}^2$. Chapter 6's operating point – $Q_{\text{g}} = 1.44 \times 10^6$ newton metres at $\omega_{\text{r}} = 1.70$ radians per second – fixes $K = 4.98 \times 10^5$, and closing that loop is what makes the pair stiff. The rotor now settles rather than drifting: it runs to the speed at which aerodynamic torque equals generator torque, $\omega_{\text{r}} = \sqrt{Q_{\text{a}}/K} = 1.736$ radians per second for $Q_{\text{a}} = 1.50 \times 10^6$, and it gets there with a time constant of

$$\tau_{\text{r}} = J/(2K\omega_{\text{r}}) = 6.0 \times 10^6/(2 \times 4.98 \times 10^5 \times 1.736) = 3.5 \text{ s} \tag{7.3}$$

Equation 7.3 is three and a half seconds against the oil's two thousand. That ratio, near six hundred to one, is exactly the appendix's definition of a **stiff** system: a fast process and a slow one in the same model, the fast one over almost immediately, so that stability rather than accuracy sets the step. An explicit solver is stable only for a step below about twice the fastest time constant – here about seven seconds – and the slow state is the one the twin actually cares about, arriving once every ten minutes with the SCADA reading – and it arrives at that rate on purpose, because the diagnostic twin of a real floating turbine chose the gearbox temperatures as its modelled variables precisely on the grounds that temperatures change slowly and need only low measurement rates [22]. So an explicit solver steps the oil rule six hundred times per reading to move it a fraction of a degree, not because the oil needs it but because the rotor would explode if it did not. The state the twin cares about is the slow one; the step is set by the fast one; and nothing about the decision justifies the cost.

7.3.3 The same ten minutes, three ways

Take one SCADA interval, $h = 600$ seconds of simulated time, from Chapter 6's state $\omega_{\text{r}} = 1.70$, $T_{\text{oil}} = 48.0$. Table 7.2 runs it three ways.

Table 7.2: One SCADA interval of the turbine's two-state model, stepped three ways. The middle row is what stiffness looks like when it is ignored: a step chosen for the quantity you care about, destroyed by the one you do not.

Way	ω_{r} after 600 s	T_{oil} after 600 s	Rate evaluations	Verdict
explicit, $h = 1$	1.736	49.04	1,200	correct, and cheap enough
explicit, $h = 600$	7.70, then diverges	49.20	2	unstable; the oil is fine and the rotor is nonsense
implicit, $h = 600$	1.735	48.92	2, plus two small equation solves	stable and usable; the rotor lands on its steady value

The middle row is worth doing by hand, because it is the failure this section exists to prevent. At $\omega_r = 1.70$ the torque imbalance is $Q_a - Q_g = 6.0 \times 10^4$ newton metres, so the rotor's rate is 0.010 radians per second per second and a six-hundred-second explicit step lands on $\omega_r = 7.70$ – a turbine spinning four and a half times too fast. Evaluate the controller there and $Q_g = K \times 7.70^2 = 2.95 \times 10^7$ newton metres, so the next rate is $(1.50 \times 10^6 - 2.95 \times 10^7)/(6.0 \times 10^6) = -4.67$, and the step after that puts the rotor at minus two thousand eight hundred radians per second. The simulation has not become inaccurate; it has become meaningless, in two steps, while returning numbers the whole way.

The implicit row solves, per step, the small equation $\omega_r = 1.70 + 10^{-4}(1.50 \times 10^6 - 4.98 \times 10^5\,\omega_r^2)$, whose positive root is 1.735 – the steady value, reached in one step, from a solver that never became unstable. The oil is even easier: the implicit step rearranges to $T_{oil} = (48 + 0.3 \times 12 + 12)/1.3 = 48.92$, against a true value of 49.04 and an explicit $h = 600$ answer of 49.20. Both are within a fifth of a degree; the choice between them was never about the oil.

7.3.4 Algebraic constraints, and the loops they make

Not every line of a model has a rate in it. Chapter 6's turbine had two that did not: the gear ratio, $\omega_g = n_{gear}\,\omega_r$, and the referral of the fast-shaft torque back to the rotor. An **algebraic constraint** is an equation that must hold at every instant, and a solver handles it by solving it after each step rather than integrating it.

That is straightforward while the constraints form a chain. It stops being straightforward when they form a cycle – when an output feeds back to its own input within one step, with no rate in between. That is an **algebraic loop**, and a solver cannot evaluate it in any order: there is no first equation. The multi-paradigm modelling literature is precise about where these come from. Equations written without saying which variables are known and which unknown – the equation-based style – have their **computational causality** assigned afterwards, and where causality cannot be assigned without conflict, the model needs a special implicit integration routine whose equations are solved by an iteration scheme inside the numerical integration method; loops of this kind are named there as zero-order causal paths [166]. In practice: your tool will either report the loop and refuse, or break it by iterating, and the second costs an inner solve per step that nobody budgeted for. If a model that should run in microseconds runs in milliseconds, an algebraic loop is the second thing to suspect, after stiffness.

7.3.5 Acausal tools, and what their translation step buys

There are two ways to write the six lines of Chapter 6's turbine. The **causal** way is the one in that chapter: a `step` function, assignments in an order you chose, the outputs computed from the inputs. The **acausal** or **equation-based** way writes the same physics as equations without choosing which side is the unknown, and lets a compiler work out the order – so a resistor is written $v = iR$ once and used in both directions, and a model is assembled by connecting components rather than by wiring signals. Modelica is the language this style is usually met in, and it is described exactly that way in the modelling literature: differential-algebraic equations written without specifying the knowns, with causality assigned afterwards to produce something executable [166].

What the translation step buys is composition: a heat exchanger written once works in any circuit. What it costs is a compilation stage between your edit and your run, and a debugging experience conducted in transformed equations rather than in the ones you wrote. In a twin the compiled artefact is the thing that ships: a model-based engineering pipeline of this kind brings Modelica and other formalisms in as compiled units with generated interface code, one pipeline producing the simulation and another the deployed system [167]. The tools also mix.

A production building tool chain couples an equation-based Modelica model of the heating, ventilation and control side to a conventional envelope simulator through the slowly-changing surface temperatures rather than the fast room air, precisely so that the interval between the two sides can be long – and long intervals reduce simulation time, particularly for the adaptive solvers that stiff systems demand [169]. Which is this section's argument in one sentence: where you put the boundary decides what step you can take, and the step decides what the run costs.

7.3.6 Consistent initialisation, every cycle

A solver cannot start from an arbitrary state. The algebraic constraints have to hold at the first instant as well as at every later one, so before the first step the simulator solves them for whatever the given state left undetermined. That is **consistent initialisation**, and in a design run it happens once and nobody notices.

In a twin it happens every cycle. The twin hands the simulator the state block's current belief – Chapter 4's `state` block, as corrected by the estimator – and asks for the next ten minutes; ten minutes later it hands over a new belief and asks again. Initialisation is therefore a **per-cycle cost**, not a start-up cost, and a simulator whose initialisation takes longer than its run is one you cannot use, however good its inner loop is. This is the second of the four criteria with a number attached to it. For the turbine's two states it is free: the gear ratio is one multiplication. For a model with a hundred algebraic constraints and a nonlinear solve among them, it is the whole budget, and the usual remedy is to keep the simulator warm between cycles and re-initialise incrementally rather than from cold.

7.4 Discrete-Event Simulation: Running the Event Queue

The execution problem of Chapter 6's **discrete-state and discrete-event** model type. The model is a set of modes and a set of things that happen; the simulator is an event scheduler; and the decision that matters is not the step size but *how many times you run it*.

7.4.1 Two ways to advance a clock

The appendix's *What a Solver and a Simulator Do* gives the loop: a **future-event list**, which is a priority queue of pending events ordered by time, and a loop that pops the earliest, jumps the clock to it, runs the handler, and pushes whatever the handler schedules. That is **next-event time advance**. The alternative is **fixed-increment time advance**: step the clock by a constant interval and process whatever fell inside it, which is what a solver does.

Figure 7.2 draws the difference on one timeline, and why the first exists is arithmetic. Chapter 6's turbine changes mode a few thousand times a year – cut-in, cut-out, curtailments, alarms, repairs. A next-event loop over a simulated year therefore executes a few thousand iterations. A fixed-increment loop at one-second resolution executes $365 \times 24 \times 3600 = 31.5$ million, of which all but a few thousand find nothing to do. The ratio is four orders of magnitude, and it is why a year of a turbine's operational life simulates in milliseconds while a year of its rotor dynamics does not simulate at all.

Fixed-increment advance is not always wrong. It is the right choice when events are dense relative to the interval, when the model is already being stepped by a solver for other reasons – the Hybrid Simulation section below – or when the output has to be sampled on a regular grid anyway. But the default for a model whose state is a mode is the queue.

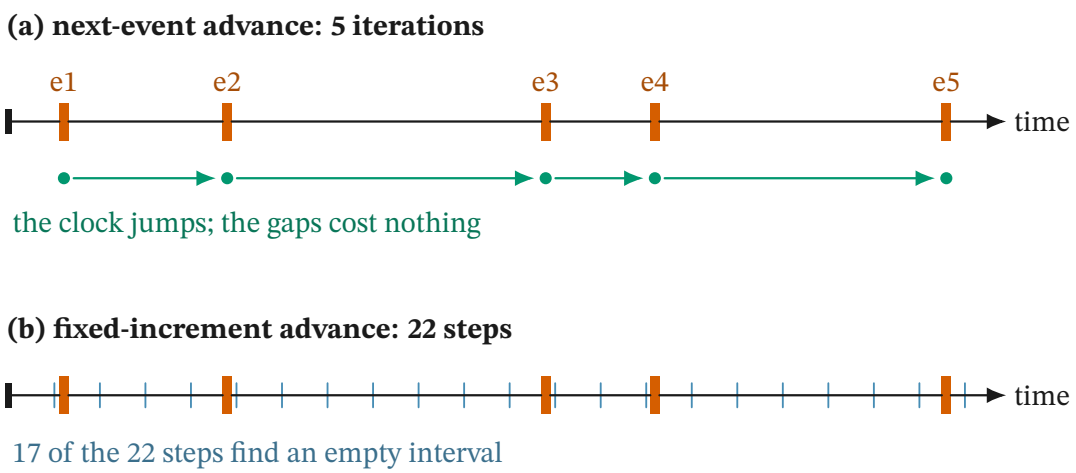


Figure 7.2: The same five events under the two ways of advancing a clock. Panel (a), next-event advance: the clock jumps to each event in turn, and the gaps cost nothing. Panel (b), fixed-increment advance: the clock steps by a constant interval and most steps find an empty interval.

7.4.2 The event formalism you will meet by its acronym

You will meet the acronym, so here is what it is. The **Discrete Event System Specification** (DEVS) is a formalism that says exactly what an atomic model owes a scheduler, and the modelling literature states it as four things: an **internal transition function** from state to state, fired when the model's own time runs out; a **time advance function** that says how long the model will sit in its current state, and whose access to the state is read-only; an **external transition function** that fires on an incoming event and sees both that event and the elapsed time since the last transition; and an initial total state [166]. Read as a software engineer: DEVS is an interface with four methods, and any model implementing it can be run by any DEVS scheduler and composed with any other. That is its whole value, and it is why a design method for twinning systems lists choosing between differential equations, DEVS and Petri nets as an explicit design stage rather than an accident of tooling [12]. This book does not teach DEVS; it names it so that the acronym on a tool's front page means something.

7.4.3 One run of a stochastic model answers nothing

This is the central lesson of the section, and it is the one most often skipped by engineers arriving from deterministic simulation.

Chapter 6's vessel-dispatch model has randomness in it: the weather windows open and close at times drawn from a winter distribution. Run its queue once and you get the five-event trace that chapter tabulated, ending in twenty hours and thirty minutes of lost production. Run it again with different weather and you get a different number. The twenty and a half hours is not the answer. It is *one sample* from the answer, and reporting it as the answer is the most common way a simulation study misleads the person who commissioned it. Computer simulation models are widely used to replicate and analyse stochastic systems, and discrete-event simulation is the most common technique for industrial settings [182] – which makes this a very widely available mistake.

Four pieces of machinery follow from it.

Random variate generation, seeds and streams. The simulator's randomness comes from a pseudo-random generator: a deterministic sequence, fixed by a **seed** s, from which variates of the distributions you need are derived. Two properties matter in a twin. First, the same seed gives the same run, which is what makes criterion 3 of the four achievable. Second, a gen-

erator can be split into independent **streams**, so that the weather uses one and the repair durations another; without that, adding a vessel changes the weather, because it changes how many numbers were drawn before the weather asked for one, and your comparison is meaningless.

Replication. A **replication** is one complete run with its own seed. The output of a stochastic model is the sample of N replications, and the answer is a summary of that sample – a mean, a spread, a percentile – with the uncertainty of the summary reported beside it.

Warm-up and run length. A model started in an empty state spends its first stretch in a condition the real system is never in: no faults pending, no vessel at sea, no backlog. Statistics collected during that stretch are biased, so the stretch is discarded as **warm-up**. The **run length** is how much simulated time each replication then covers, and it is chosen so that the quantity of interest has been exercised enough times to matter – for the turbine, a whole winter, because the whole question is about winter.

Confidence intervals. Report the answer as an interval. For a mean over N independent replications with sample standard deviation s_N, the usual ninety-five per cent interval is

$$\text{mean} \pm 1.96\, s_N/\sqrt{N} \tag{7.4}$$

and the half-width in Equation 7.4 is the number to quote, not the mean alone.

7.4.4 The vessel run, a thousand times

Chapter 6's five-event trace, run a thousand times with the weather windows drawn from their winter distribution rather than fixed. Suppose the sample comes out with a mean lost production of 20.4 hours per fault and a sample standard deviation of 6.0 hours – a spread that wide because the waiting-for-a-window part, nine and a half hours in that chapter's single trace, is sometimes forty minutes and sometimes three days.

The ninety-five per cent interval on the mean is then $1.96 \times 6.0/\sqrt{1000} = 0.37$ hours either side: 20.4 hours, plus or minus about twenty-two minutes. If the planner wants the mean pinned to half an hour – a quarter of an hour either side – rearrange Equation 7.4:

$$N = (1.96\, s_N/\text{half width})^2 = (1.96 \times 6.0/0.25)^2 = 2213 \tag{7.5}$$

Two thousand two hundred and thirteen replications, which at a few milliseconds each is a second of compute. Equation 7.5 is the arithmetic behind every "how many runs?" question in this chapter, and its shape is the shape of the whole subject: the half-width falls as one over the square root of N, so **ten times the precision costs a hundred times the runs**.

Two warnings the arithmetic does not carry. First, it is the interval on the *mean*. If the decision is about the bad winters – the ninety-fifth percentile of lost production, which is what a spares policy is sized on – the same precision needs substantially more replications, because a percentile out in the tail is estimated from far fewer effective observations than a mean. Second, the interval says how well you know *this model's* mean. It says nothing about whether the model's weather distribution resembles the North Sea. That is the credibility chapter's question, and no number of replications answers it.

7.4.5 Trace-driven replay

The other way to run the queue is not to generate the events at all. **Trace-driven replay** feeds the scheduler the control room's own event log – last winter's actual alarms, dispatches and windows – and runs the model against them. It answers a different question: not "what will happen", but "what would this policy have done to what did happen". It needs no distributions, which removes the largest modelling assumption, and it is limited to the history you have, which

is usually one or two winters and no rare events at all. Use it to check that a model reproduces a known year before trusting its generated ones; that is the cheapest verification in this chapter.

Replay is also what makes the event queue a component of a what-if loop rather than a study. In a published production twin the discrete-event model is called repeatedly by an optimiser: candidate schedules are generated, each evaluated by the discrete-event model of the production system, the best by makespan are kept, a new population is generated, and the loop runs until a stopping criterion is met, after which the winning sequence is fed back to the operator or to the manufacturing execution system [183]. The simulator there is not answering a question; it is being called a few thousand times as a subroutine, which is a workload, and the Driving a Simulator from a Twin section below names it as a run mode.

7.5 Hybrid Simulation: Where the Two Meet

Chapter 6 named the **hybrid automaton** – a state machine whose modes each carry their own rate rules – and stopped, handing the execution question here. It is a real question, and it has three mechanisms and three costs.

A turbine in start-up obeys one set of rate rules, a turbine producing under torque control another, and a turbine in fault with the brake on a third. The mode machine chooses which rate rules the solver runs; the solver's states decide when a guard fires. Two loops, each driving the other. The manufacturing-twin literature's own table lists the hybrid automaton among the model types used for machine and process states, with a generic hybrid automaton written for the continuous and discrete-event dynamics of additive processes [91]; this section is what running one costs.

7.5.1 Zero-crossing detection

A guard is a condition – the rotor speed reaching rated, the oil passing an alarm temperature. Between two solver steps, a condition is either true or false; the switching **instant** is somewhere inside a step and the solver walked straight past it.

The remedy is to write each guard as a function that changes **sign** at the switching point – $\omega_\mathrm{r} - \omega_\mathrm{rated}$, rather than $\omega_\mathrm{r} \geq \omega_\mathrm{rated}$ – and have the solver watch it. At the end of each step the solver checks whether any guard function changed sign; if one did, it does not accept the step. It searches inside the interval for the instant of the crossing, usually by bisection or interpolation, truncates the step there, applies the transition, and starts again.

The cost, in Figure 7.3, is the search – a handful of extra rate evaluations per switch – and the benefit is that the switch happens at its time. Skipping it is tempting and occasionally defensible: if your step is a millisecond and your guard fires once a minute, the error from switching at the end of the step is a millisecond of the wrong physics. If your step is ten minutes, it is ten minutes of the wrong physics, and the alarm the model should have raised happens in the wrong state. This is one reason a twin that has to detect mode changes cannot take the biggest step its accuracy allows, and it is precisely the case adaptive solvers advertise: short steps where the dynamics are fast **and where the instant at which something happens must be found properly** [169].

7.5.2 State reset, and the restart it forces

A transition may change the continuous state as well as the mode. A fault trip applies the brake and sets the generator torque to zero; the rotor's speed does not jump, but the torque driving it does, so the *rate* is discontinuous. A contact model, or a tank that is suddenly isolated, can make the state itself jump.

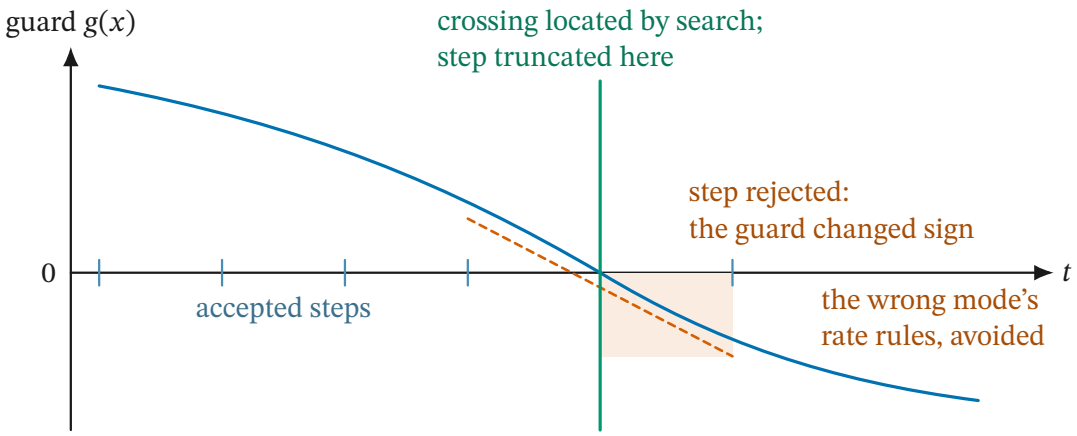


Figure 7.3: Locating a switching instant inside a step. The solver takes a step, notices that the guard function changed sign, searches inside the interval for the crossing, truncates the step at it, applies the transition, and restarts from there. Without the search, the model runs the wrong mode's rate rules for the shaded fraction of the step.

Either way the solver's accumulated history is now wrong. An adaptive solver has spent the last few hundred steps learning how smooth this trajectory is and how long a step it can afford; a multi-step method has several past points it is extrapolating from. At a switch, all of it is discarded: the solver **restarts** from the new state with no history and a conservative first step, and climbs back to its cruising step size over the next several steps.

That is the real cost of a hybrid model, and it is counted in switches, not in steps. The declarative-modelling literature states it plainly: with a variable-step solver, events influence the solver's behaviour because integration must stop every time an event is triggered, and a high number of events slows the simulation down [181]. A model with a thousand switches an hour is not a model with a thousand extra evaluations an hour; it is a model whose solver never gets to take a long step.

7.5.3 Chattering, and the hysteresis band

The pathological case is a guard that fires, is undone by its own transition, and fires again. A thermostat whose switching threshold is exactly its set point switches every step. A curtailment rule that compares an instantaneous power to a limit switches on every gust. The solver detects a crossing, truncates, transitions, and immediately detects the reverse crossing: the simulated clock advances by microseconds per iteration and the run never finishes. This is **chattering**, and its arrival is unmistakable – wall-clock time passing with simulated time standing still.

The ordinary remedy is a **hysteresis band**: switch up at one threshold and down at a lower one, so that a return trip requires a real change rather than a wobble. Mode changes then cost a minimum amount of movement, which is also true of the physical asset, so the band is usually a better model as well as a runnable one. The alternatives – a minimum dwell time in each mode, or filtering the guard's input – do the same job in the time domain.

7.5.4 The turbine through a curtailment and a trip

Chapter 6's six-mode machine driving the two-state rotor-and-oil model, with an adaptive solver at a cruising step of five seconds and a restart costing the solver about ten steps to climb back.

A curtailment. The grid operator caps output at noon and lifts the cap at 16:00. Two guards fire, both on an input rather than on a state, so no search is needed; the mode changes, the

commanded torque changes, neither state jumps. Two restarts, twenty steps of climb-back, and nothing else. Over four hours at a five-second step that is 2,880 steps, of which 20 were slow: under one per cent overhead.

A fault trip. A gearbox alarm at 02:10 trips the turbine. The guard is on a state – an oil temperature crossing a limit – so the solver searches inside a step for the crossing instant. The transition applies the brake, which sets Q_{g} to the brake torque and Q_{a} to approximately zero: the rate is discontinuous, the solver restarts, and the rotor's dynamics are now much faster than they were, so the solver's first attempt at its old step is rejected and it takes several tries to settle. Call it one search and one restart costing thirty steps. Then the turbine sits in fault for twelve hours with the rotor stopped and only the oil cooling – during which the adaptive solver stretches its step to minutes, because nothing is happening, which is the same arithmetic as the event queue arriving at the solver.

The lesson to keep is the arithmetic of the two together: a day with two curtailments and one trip costs three searches and four restarts, perhaps seventy steps of overhead against seventeen thousand steps of work. A day of a chattering guard costs everything. The number of switches, not the number of steps, is the quantity to watch.

7.5.5 The mode belongs in the state record

Chapter 6 asked for this and this chapter can now say why. The mode is part of the state the simulator is restarted from, so it must be part of the state the twin hands it. A simulator restarted with the right temperatures and the wrong mode will run the wrong rate rules from the first step, and produce a smooth, plausible, completely wrong trajectory – the failure mode hardest to notice, because nothing crashes. Put the mode in the state block beside the speed and the temperature, and put its `as_of` beside it too.

7.6 Stochastic Simulation: Many Runs Instead of One

The execution problem of Chapter 6's **probabilistic and Bayesian** model type. It comes before the field-solver section, out of the models chapter's order, for one reason: the field section's verdict is that you cannot afford to run the field model, and the argument for what to do instead needs the ensemble already on the table.

The previous section met randomness inside one model and answered it with replications. This section is about the case where the *uncertainty itself is the output*: the twin is asked how long the bearing has, and the honest answer is a distribution.

7.6.1 The ensemble

Chapter 6 modelled the bearing's wear as a Wiener process with drift,

$$W(t+h) = W(t) + \mu\, h + \sigma\sqrt{h}\,\xi, \qquad \xi \sim \mathcal{N}(0,1) \tag{7.6}$$

and Equation 7.6 is the step this section runs many times. That chapter then used one month's vibration reading to update a belief over the drift μ across three candidates – 0.5, 1.0 and 2.0 millimetres per second per month – arriving at a posterior of 0.36, 0.53 and 0.11. That posterior is where this section starts. The execution recipe is four lines:

```
for i in 1 .. N:
    mu_i = draw from the posterior over mu        # parameter uncertainty
    W = W_now                                     # the state block's belief
    t = 0
```

```
while W < W_f:
    W = W + mu_i * h + sigma * sqrt(h) * normal(0, 1, stream=i)
    t = t + h
record t                                  # this member's crossing time
```

Each pass is one **ensemble member**; the recorded crossing times are the sample; their histogram *is* the remaining-life distribution. Figure 7.4 draws it.

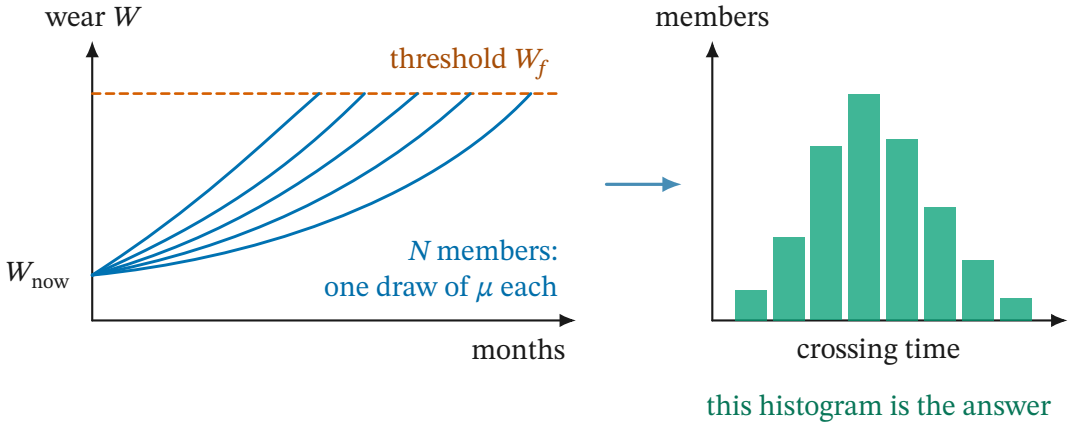


Figure 7.4: An ensemble as a workload. Left: N trajectories started from the same state, each with its own draw from the parameter posterior and its own random path, run until they cross the failure threshold. Right: the crossing times collected into the histogram that is the answer. One trajectory is not a small version of this picture; it is a single bar.

Two things to notice about the loop. The draw of `mu_i` happens **once per member**, outside the time loop: it is a parameter, not a noise term, and re-drawing it each step would model a bearing whose wear rate changes every month rather than a bearing whose wear rate is unknown. And the two sources of spread are different in kind – the draw of `mu_i` is what the twin does not know about this bearing, the ξ term is the wandering the bearing will really do – which is why the ensemble has to carry both and why a twin that reports only one of them is reporting the wrong confidence.

Chapter 6's three-centre mixture is this ensemble with $N = 3$ and the random term switched off: three candidate drifts, three crossing times, weighted by the posterior. That is a perfectly good answer when the parameter has three candidates. It stops being one the moment μ is continuous, the model has three uncertain parameters instead of one, or the threshold depends on the operating condition – at which point the only tractable method is to sample.

7.6.2 How big is N

The same arithmetic as the last section, with the same shape. If the ensemble's crossing times have a spread s_N and the answer wanted is the mean, the ninety-five per cent half-width is $1.96\, s_N/\sqrt{N}$, so ten times the precision costs a hundred times the runs.

Work one. Suppose the bearing's ensemble gives a mean remaining life of 14.2 months with a spread of 4.8 months. At $N = 100$ the half-width on the mean is $1.96 \times 4.8/10 = 0.94$ months – the twin knows its own average to within four weeks, which is useless when the maintenance window is monthly. At $N = 2000$ it is 0.21 months, about six days, which is inside the window. Two thousand members, each a few thousand steps of four arithmetic operations, is a few tens of millions of operations: milliseconds. For a bearing, N is free and there is no excuse for a small ensemble.

N is free because the step is four operations. The moment the step is a field-solver run, the same N is a supercomputer-month, and that is the trade the next two sections are about.

One caution, the same one the vessel run carried. The decision is rarely about the mean. "Schedule the vessel so that there is a five per cent chance of failing first" asks for the fifth percentile of the crossing times, and a tail percentile is estimated from the handful of members that landed out there, so it needs far more members than the mean does for the same precision. Decide which statistic the decision needs *before* sizing the ensemble.

7.6.3 Variance reduction and common random numbers

Most ensembles in a twin exist to compare two options, not to measure one. Does a second vessel pay for itself? Does replacing the bearing now beat replacing it in six months? The quantity wanted is a **difference**, and a difference between two noisy estimates is noisier than either.

Common random numbers is the cheapest fix in simulation, and it is one line of code: run both options with the *same* seeds, so that member i of option A and member i of option B see the same weather, the same arrival times, the same wear kicks. The randomness then cancels out of the difference, and what is left is the effect of the decision. Run them with different seeds and you may need ten times the members to see the same effect. The practical requirement is the independent streams of the previous section: option B must not consume a different number of random draws before reaching the weather, or the runs are no longer paired.

Other variance-reduction techniques exist – antithetic variates, control variates, importance sampling for rare events – and all of them are worth reaching for before reaching for more compute. Importance sampling in particular is how a question like "what is the chance of three simultaneous faults in a winter" is answered at all, because a plain ensemble would need millions of members to see the event once.

7.6.4 Seeds, determinism, and a run you can repeat

Criterion 3 of the four said a simulator must be deterministic enough to be compared with itself, and an ensemble is where that criterion is either met or quietly lost. The requirement is precise: given the same model version, the same initial state, the same inputs and the same seed, the run returns the same numbers.

Three things break it in practice, and all three are software problems rather than numerical ones. An unseeded generator – seeded from the clock, or from the process id – makes yesterday's prediction irreproducible for no gain. Parallel execution that assigns members to workers dynamically makes the *set* of members reproducible but not their order, which is harmless for a histogram and fatal for a paired comparison; give each member its own stream indexed by i rather than drawing from a shared generator. And floating-point summation whose order depends on thread scheduling makes bitwise comparison impossible; usually tolerable, occasionally not.

Beyond the seed sits the environment. Computational reproducibility is defined as obtaining consistent and verifiable results from a computational experiment when the same input data, code and software environment are used, and containers are the standard way to pin the third of those – packaging a library with its dependencies so that another party can run it without a dependency conflict and reproduce the result [184]. The twin literature makes the same point about itself: reproducibility means a different team obtaining the results with the same experimental setup, replicability a different team with a different setup, and the reason so few twin results are either is that reproducing the physical side of the experiment is prohibitively expensive – which is why affordable, fully published exemplars are built on purpose [31]. For a twin's simulator the practical reading is short: the seed goes in the provenance record, and so does the container digest.

7.6.5 When the posterior is expensive

Chapter 6 flagged where this section ends up. Its three-candidate table was a teaching simplification; a real twin carries μ as a continuous distribution, and updating a continuous posterior is done by sampling – the twin review lists Markov chain Monte Carlo, Gibbs sampling and slice sampling as the methods [11]. Sampling means evaluating the model once per proposal, and a chain is thousands to millions of proposals long.

That arithmetic is what puts a surrogate inside the loop, and the structural-health literature shows the assembled result: a multi-fidelity neural surrogate maps damage and operational parameters onto sensor recordings and is called by a Markov chain Monte Carlo procedure to update the distribution of the structural state from noisy observations, standing in for a finite-element discretisation of several thousand degrees of freedom that the chain could not have afforded to call [174]. The same shape appears in clinical twins, where the output for a patient is a family of posterior trajectories rather than one forecast, and the quantified uncertainty is what makes the downstream decision risk-aware rather than merely optimal [185].

Hold that thought for two sections. First, the model the surrogate is standing in for.

7.7 Field Solvers, and Why They Live Outside the Loop

The execution problem of Chapter 6's **high-fidelity field** model type. The answer, arrived at honestly, is that there is no execution: this model type cannot run inside a twin's decision loop for any purpose, and the value of the section is in seeing exactly why and what the field model is therefore *for*.

To add physical realism to a twin the governing equations of the physics have to be solved, and for most real geometries no analytical solution exists [186]. Solving them numerically is a four-stage pipeline, and each stage has a cost a software engineer should be able to recognise.

7.7.1 Meshing, and the mesh convergence study

The body is divided into elements. The mesh is not an input to the answer; it is part of the model, and a coarse mesh gives a different answer from a fine one. Which raises the question the whole discipline turns on: is this answer the model's, or the mesh's?

The **mesh convergence study** is how that is settled, and it is the one piece of field-solver practice that belongs in a twin engineer's vocabulary. Solve on a mesh. Refine it – halve the element size, or double the element count – and solve again. Repeat until the quantity you care about stops moving by more than your tolerance. The mesh at which it stopped moving is the mesh you use; the ones before it produced answers about discretisation. The study is a fixed cost paid once per model, before any twin runs, and it is a cost that has to be repaid every time the geometry changes.

It is also the cleanest example in this chapter of a **verification** check as against a validation one. A converged mesh says the answer is the model's. It says nothing whatever about whether the model resembles the turbine.

7.7.2 Assembly and the sparse solve

The elements are assembled into a system of equations – one unknown per degree of freedom, a million of them for a tower and blades – and the system is solved. The appendix's *Dimension, and cost growing with it* costs a dense step of a million-state model at around 10^{12} multiplications and notes that a real solver exploits the zeros: the matrix of a meshed body is **sparse**, because an element's equation involves only its neighbours, and everything in the field is built on that.

Two families do the solve. A **direct** solver factorises the matrix once and then solves cheaply for many right-hand sides; it is predictable, and its memory use is the thing that stops it. An **iterative** solver improves a guess until the residual is small enough; it uses far less memory, and its convergence depends on the problem – which is what **preconditioning**, transforming the system into one that converges faster, exists to fix. Choosing between them and tuning a preconditioner is specialist work. Knowing that the choice exists, and that it is where a run's time actually goes, is not.

7.7.3 Eigenvalue extraction is a different, cheaper run

This is the detail Chapter 6 used without naming. That chapter compared the tower's *computed* first sway frequency against the one measured by operational modal analysis. Computing a natural frequency is not a time-domain simulation at all. It is an **eigenvalue problem** – find the shapes in which the structure prefers to oscillate and the frequencies at which it does so, which the appendix's *Modes* section describes – and extracting the first few modes is far cheaper than integrating the structure's motion through time, because it asks a question about the structure rather than about a particular loading.

That is why structural twins are built on it. Measured natural frequencies from operational modal analysis serve as the target values for a parameter optimisation of a detailed finite-element model's design variables, which is finite-element model updating [172]; the parameters such an update adjusts are the mass, centre of gravity, element stiffness, local joint stiffness, soil stiffness and damping, driven by a sensitivity matrix until the analytical modal parameters match the experimental ones [165]. And once updated, the same eigenvalue run yields what no sensor can: an aircraft twin's wing model, calibrated on the first two bending modes extracted from strain data, then computes higher bending, torsional and skin-buckling modes that are hard to characterise experimentally [171].

So the field model's most useful run, inside a twin, is not a simulation. It is an eigen-analysis on a schedule.

7.7.4 Where it actually runs

Batch. On a cluster or a large machine, on a schedule – quarterly, after a storm, after an inspection – or on a trigger, off the critical path entirely, with results delivered to the twin as a finished artefact rather than as a service call. A run takes hours, sometimes days: the offshore structural-health review says exactly that, adding that understanding a structure's behaviour requires *repeated* simulations with varying input parameters, which multiplies the cost again [40].

Two operational consequences follow from "hours to days" that a software engineer owns rather than the analyst. **Checkpoint and restart**: a twelve-hour job that cannot resume from an interruption will, on a shared cluster with a wall-clock limit, never finish. And **job scheduling**: the twin's request for a field run is a submission to a queue, so the twin's interface to this model type is asynchronous by construction – submit, poll, collect – and pretending otherwise is how a twin's architecture acquires a synchronous call that blocks for six hours.

7.7.5 The verdict

The appendix computes the number. A million-state field model whose step is 10^6 operations, stepped at the resolution the physics requires, comes out at a real-time factor of about one hundredth: a hundred days of compute for a simulated day.

A factor of one hundred is not closed by buying a faster computer, and it is not closed by waiting. It rules the field model out of the loop for every value pattern in Chapter 1 – not only

control, but monitoring and diagnosis too, because even the ten-minute SCADA cycle is a thousand times faster than this model can answer. The unit-level manufacturing review states the general case in one sentence: high-fidelity numerical models are not always appropriate for twin applications, because fidelity comes at the price of computational cost or simulation time, and reduced-order modelling or simplification is what is used instead [91]. Reduced-order modelling is named as a key enabler for compressing high-fidelity models into much lower dimensions precisely to relieve that demand [186].

Which is the next section. The field model stays – it is what the reduced model is made from, and what the eigenvalue run updates – but it stays outside, in batch, and something else goes in the loop.

7.8 Making an Expensive Simulator Cheap

The execution side of Chapter 6's **reduced-order and surrogate** model type. That chapter said what the cheap model *is*. This section says what it costs to get one, and the argument is a single sentence: **the cheap model is bought with an expensive simulation campaign.** Figure 7.5 puts the two halves of that bargain side by side: an offline half that runs once and costs weeks, and an online half that runs every cycle and costs microseconds.

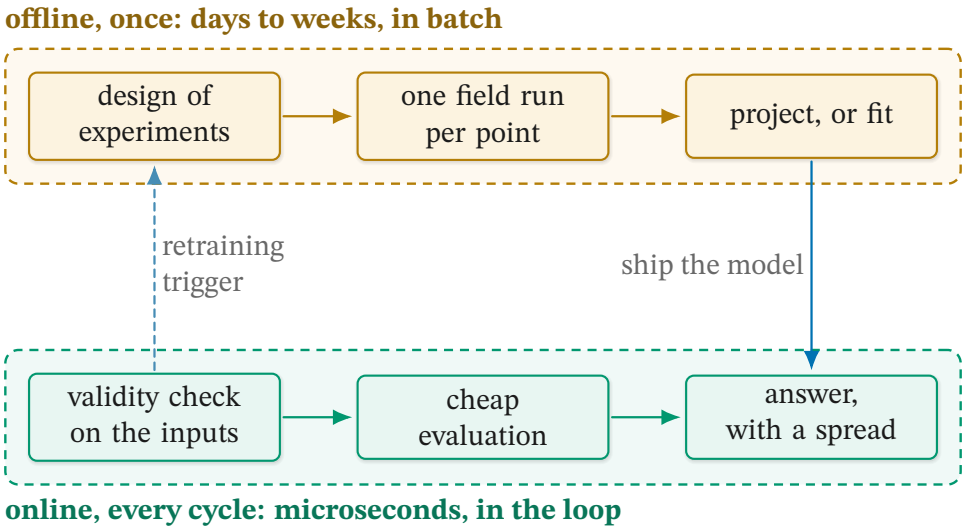


Figure 7.5: The offline campaign and the online model. Offline, once: a design of experiments over the input ranges the asset will meet, one field-solver run per point, the runs collected and either projected onto a basis or fitted to a function. Online, every cycle: the cheap model, guarded by a validity check that refuses inputs outside the region the campaign covered, with a retraining trigger back to the offline side.

7.8.1 The campaign, priced

The surrogate is a function fitted to the field model's own output, so before it can be fitted, the field model must be run – once per training point. Choosing those points is the **design of experiments**, and it is where the money goes.

Work the arithmetic. Suppose the tower's fatigue surrogate of Chapter 6 takes five inputs: wind speed, turbulence intensity, wave height, wave period and yaw misalignment. Sample each at five levels on a full factorial grid and the campaign is

$$5^5 = 3125 \text{ runs} \tag{7.7}$$

At four hours a run that is 12,500 core-hours, or about 521 days on one core; on sixty-four cores, eight days of wall-clock time and a noticeable cloud invoice. And five levels per input is coarse: it resolves the response into four intervals per axis, which for a nonlinear quantity like fatigue damage is not much.

Equation 7.7 is why a grid is usually the wrong design. The count grows as levels to the power of inputs, so the sixth input multiplies the campaign by five. The usual answers are to sample the space rather than grid it – a Latin hypercube spreads a fixed budget of points evenly over every axis at once – or to sample adaptively, fitting after an initial batch and spending the next runs where the fit is worst. Both turn "how many levels" into "how many runs can I afford", which is the question with a budget attached to it.

This is the step the literature names as a first-order obstacle rather than a detail: the prohibitive cost of generating sufficient data for surrogate model training is listed among the foundational challenges for twins [90], and the structural-health review, having noted that one run takes hours or days, identifies the availability or generation of training data sets as the main problem with substituting a surrogate at all [40].

7.8.2 Projection: the basis you already paid for

For the **reduced** model rather than the fitted one, there is a shortcut, and Chapter 6's tower is the example. The reduction projects the million-state model onto a handful of shapes and keeps only the coefficients of those shapes. Where do the shapes come from? From the modes – and the modes are what the eigenvalue run of the previous section already computed, for the model-updating loop.

So the reduced tower is very nearly free once the field model exists and has been updated: the expensive part, the eigen-analysis, was already on the schedule for a different reason. That is the cheapest path to a fast model in this chapter, and it is available whenever the quantity of interest is dominated by a few modes – structures, acoustics, thermal diffusion. It is not available when the behaviour is not modal, which is when you are back to fitting.

The literature's mature form of this is a **library**: physics-based reduced-order models built component by component from high-fidelity finite-element simulations, which scale to a large asset and, shared across assets, to a fleet of twins, with a classifier inferring from sensor data which library models best describe the asset now – the motivation being stated exactly as this section's: large-scale physics-based models are intractable in a real-time or many-query context [66]. Reduced models built to be interpretable and reusable across processes are argued for on the same grounds in manufacturing [173], and the executable-twin literature reports the finished article: a detailed finite-element model of a turbine blade compressed into a reduced-order model which, with a state estimator, shows the full strain field in real time from a handful of strain gauges [20].

7.8.3 Fitting, and the multi-fidelity trick

For the **surrogate**, the campaign's outputs are training data and the surrogate is whatever function you fit to them – a response surface, a Gaussian process, a kriging model. Chapter 6 used a Gaussian process for the fatigue surrogate because it returns a spread alongside its prediction, which is the property that makes the validity check below possible. Surrogates are defined in the twin literature as exactly this: cheaper approximations that mimic computationally expensive physics models, built from training data taken from the original simulation, not a twin in

themselves but usable inside one when the original model is too slow for the decision interval [92].

The **multi-fidelity** trick halves the campaign. Run a few points on the expensive model and many on a cheap one – a coarser mesh, a smaller domain, a simplified physics – and fit a surrogate that learns the *correction* from cheap to expensive rather than the expensive function from scratch. The structural-health surrogate cited in the Stochastic Simulation section is built this way, and its purpose is precisely to survive being called thousands of times by a sampler [174]. When the cheap model is ten times faster and the correction is smooth, the campaign cost falls by most of an order of magnitude.

7.8.4 The validity check, as code

Chapter 6 gave the warning: a surrogate is valid only where it was trained. This section turns it into something a simulator executes, because a warning in a book does not stop a twin extrapolating on a February morning.

```
def predict(surrogate, u, run_log):
    """Evaluate the surrogate, or refuse. Never extrapolate silently."""
    if not surrogate.domain.contains(u):
        run_log.record("surrogate: input outside training domain", u)
        return Refused(reason="outside training domain",
                       fallback="field model, batch")
    mean, spread = surrogate.evaluate(u)   # the process returns both
    if spread > surrogate.spread_limit:
        run_log.record("surrogate: spread above limit", u, spread)
        return Uncertain(mean, spread)     # usable, but flagged
    return Ok(mean, spread)
```

Three properties of that listing are the point of it. The domain check is on the **inputs**, before any evaluation, and is a box or a hull recorded by the campaign – not a guess. The spread check is the surrogate's own admission of ignorance, which is why the fitted model was chosen to produce one. And the refusal is a **return value**, not a log line: the caller has to decide what to do without an answer, which is the only way the twin's behaviour outside the training region is designed rather than discovered.

7.8.5 The retraining trigger, and the offline/online split

Two rates, and they are the reason this section is structured the way it is.

Offline, rarely: the field model, the mesh convergence study, the campaign, the fit. Days to weeks, in batch, off the critical path, and repeated only when something makes the campaign stale.

Online, every cycle: the surrogate evaluation and the validity check. Microseconds, in the loop.

The **retraining trigger** is what joins them, and there are three honest ones. The validity check refuses, or flags high spread, on a growing fraction of cycles – the asset has moved into a corner the campaign did not cover. The field model has been updated, so the surrogate now mimics a model that no longer exists. Or the asset itself has changed – a blade replaced, a foundation grouted – which is a configuration change and the configuration and manager chapter's business, but it invalidates the campaign all the same. The National Academies' report names the obligation created by all three: surrogates that make dynamic adaptation feasible must themselves be updated and validated as the representation evolves, and it lists this among the open

research gaps rather than among the solved problems [90]. A twin that ships a surrogate without a retraining trigger has bought a fast model and a slow-growing lie.

7.9 Time: Simulated Clock, Wall Clock and the Twin's Clock

Every section so far asked whether a model *can* be run. This one asks whether it can be run **here**, and it is the pivot of the chapter.

The appendix defines the **real-time factor** as simulated time covered per unit of wall-clock time,

$$RTF = t_{\text{sim}}/t_{\text{wall}} \tag{7.8}$$

and that is the last time this chapter defines it. Equation 7.8 is a pure number, and the three regimes it falls into, drawn on one axis in Figure 7.6, are three different jobs.

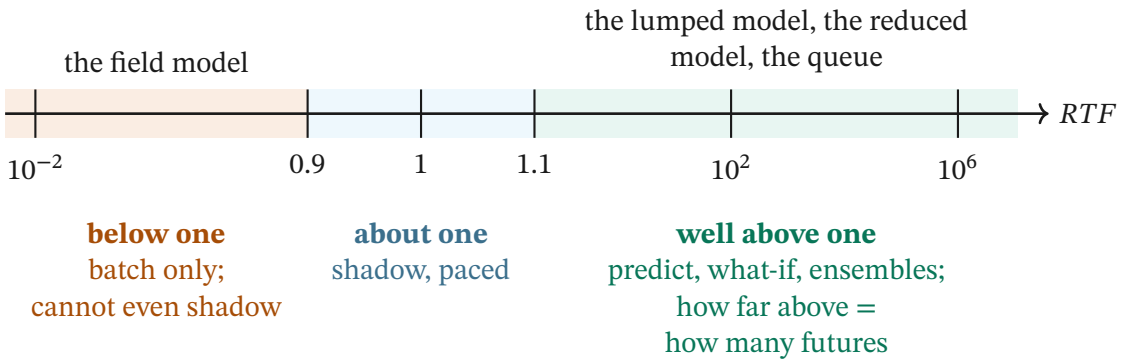


Figure 7.6: The three real-time regimes on one axis. Below one, the simulator falls behind the world and can only be run in batch, after the fact. Around one, the simulator is paced to consume readings as they arrive and shadows the asset. Above one, the simulator outruns the world, and how far above decides how many futures fit inside a decision's latency.

7.9.1 Three regimes

RTF **below one: batch only.** The simulator cannot keep up with the world, so it can only ever be run after the fact, on a horizon that has already happened or on a schedule. The field model of the previous section, at $RTF = 0.01$, is here, and nothing about the deployment changes that. Note what it rules out: not merely control, but *shadowing*. A model that cannot cover a minute of asset behaviour in a minute cannot be kept in step with the asset at all.

RTF **about one: shadow, and pace.** The simulator runs alongside the asset, consuming readings as they arrive and producing its expectation as the asset produces its reality. Nothing in this regime happens by accident: a simulator whose natural *RTF* is 50 must be **paced**, or it will run fifty minutes of simulated time in the first minute and then sit idle with nothing to consume. Pacing is the engine deliberately sleeping – compute the next step, note the wall-clock time it should be delivered at, wait. The twin of a floating turbine reported in the literature is built for exactly this regime, intended to run in real time on a cloud platform beside its asset [164].

RTF **well above one: predict and explore.** This is what makes a twin more than a shadow. At $RTF = 10$ the simulator covers ten minutes of future in a minute, or ten one-minute futures; at $RTF = 1000$, a thousand. The figure that matters is not *RTF* alone but *RTF* against the decision's latency: a decision with a five-minute budget and a simulator at $RTF = 600$ can afford 600 ×

$5/60 = 50$ hours of simulated future, which can be one fifty-hour forecast or fifty one-hour what-ifs. That division is the single most useful calculation in this chapter, and it is why faster-than-real-time simulation is reported as an enabling property rather than a nicety – the modelling survey cites work using a faster-than-real-time twin to predict the dynamic behaviour of smart grid systems [186].

7.9.2 Pacing, and what to do when it fails

Pacing costs nothing while there is slack. The interesting case is when there is not: the simulator is asked for a ten-minute horizon every ten minutes, and this cycle takes eleven. A twin has four options and they should be chosen in advance rather than discovered.

- **Shorten the horizon.** Predict five minutes instead of ten. Cheapest, and it degrades the answer in a way the consumer can be told about.
- **Coarsen the model.** Switch to the surrogate, or a larger tolerance. Degrades accuracy rather than reach.
- **Drop the cycle.** Skip this prediction and re-initialise from the next state. Acceptable for a prediction, never for an accumulator.
- **Fall behind.** The worst option, and the default if the other three were not implemented: the twin's answers arrive later and later, describing a world that is further and further in the past.

7.9.3 Time discrepancy, and getting back in step

Chapter 5 defined the time discrepancy $D = t_i - t_e$ between the instant a twin holds and the instant its asset is at. This chapter reads the same quantity as how far a *running simulation* has fallen behind its asset, and the published measurements are worth quoting in numbers because they are more sobering than the definition.

In a study that set out to measure and address exactly this, the maximum observed discrepancy during a deliberately degraded period was around five seconds; when normal conditions returned, the twin caught up with wall-clock time in 60.8 time steps of 100 milliseconds, averaged over five runs – about six seconds of catching up for five seconds of falling behind, and possible only because the twin could run faster than real time [141]. Two findings from the same work matter more than the headline. First, **the maximum step a model accepts bounds how fast it can catch up**: a component with a stricter limit on its step size slows the pace at which the whole twin gets back in sync, so a model chosen for accuracy can prevent resynchronisation. Second, **the threshold was exceeded on occasion even during a normal run**, which the authors read as a need to estimate the threshold from the variability of normal operation rather than from an ideal. A twin whose "behind schedule" alarm was set from a quiet afternoon will cry wolf.

Chapter 5's twinning rate is the other side of this: the rate at which the twin is synchronised with its asset, which the manufacturing review is careful to distinguish from the rate at which the data is sampled [91]. A simulator sets an upper bound on the twinning rate that no amount of data can raise.

7.9.4 How far ahead is honest

RTF says how far ahead you *can* run. It does not say how far ahead you *should*, and the two are not related.

The appendix's *State Machines with Probabilities* has the argument in its cleanest form. Predict a Markov chain forward without looking at the world and the belief spreads at every step,

drifting towards the chain's long-run distribution – at which point the prediction carries no information about *this* asset at all, only about assets of this kind. Certainty leaks away with every step predicted without a correction, and it leaks at a rate set by the system, not by the simulator. **A prediction horizon is therefore set by how fast the belief decays, not by how fast the simulator runs.** A simulator at $RTF = 10^6$ can produce a thirty-year forecast of the turbine's rotor speed in a second; nobody should read it.

The practical form of the rule: run ahead to the horizon at which the predicted spread reaches the width at which the decision flips, and stop there. For the bearing's remaining life that horizon is months, because the wear model's spread grows slowly. For the rotor under a gusting wind it is seconds.

Table 7.3 ties the two clocks to Chapter 5's age of twin.

Table 7.3: What each of the turbine's models can sustain. Read the last column as the smallest age of twin the simulator does not prevent – the data usually decides, and the field model is the one row where the simulator decides instead.

Model, on the turbine	Achievable *RTF*	What that allows	Age of twin it can sustain
two-state lumped, implicit, $h = 600$	about 10^7	prediction, what-if, ensembles, anything	bounded by the data, not the simulator
mode machine and vessel queue	about 10^8 for a simulated year	a thousand-member ensemble in a second	bounded by the event log's latency
bearing Wiener ensemble, $N = 2000$	about 10^6	a fresh remaining-life distribution every cycle	monthly readings; the simulator is never the constraint
reduced tower, three modes	about 10^3	load estimates every second, with headroom	one second
fatigue surrogate	about 10^9 per evaluation	ten-minute fatigue totals for microseconds	ten minutes
field model of tower and blades	about 10^{-2}	a quarterly batch run	not applicable; it is not in the loop

The shape of Table 7.3 is the shape of the whole chapter: five of the six rows are limited by the data arriving, not by the arithmetic, and the sixth is limited so badly that it is not in the loop at all. That is a comfortable place to be, and it is a consequence of Chapter 6 having chosen small models for the decisions that needed speed.

7.10 Driving a Simulator from a Twin

Everything above is about a simulator. This section is about the twin calling one, and it is the section to copy from.

7.10.1 The call

One function, four things in, two things out.

```
def run(model_ref, initial_state, inputs, horizon, *, mode, seed, solver):
    """Advance a model from a state the twin supplies, and say what was run.

    model_ref     which model, and which version of it
    initial_state the state block's current belief, with its as_of
    inputs        the input series over the horizon: measured,
                  forecast or assumed
    horizon       how far ahead, in simulated time
    mode          one of the five run modes below; fixes where
                  the inputs come from
    seed          fixes the run's randomness; recorded, never
                  defaulted to a clock
    solver        the step kind and its settings

    returns (outputs, provenance)
    """
```

Four things about that signature are deliberate and each corresponds to a criterion from the What a Simulator Is section.

`initial_state` **is an argument, not a constructor parameter.** The simulator is restarted from the twin's belief every cycle – criterion 2 – so setting the state is a normal call, not a set-up phase. If your simulator can only be initialised at construction, you will pay to rebuild it every cycle, and the Consistent Initialisation subsection above is where that cost lives.

`inputs` **is a series over the horizon, not a value.** The twin must supply the wind, the load, the grid command for the whole interval being simulated. Where that series comes from is what distinguishes the run modes below, and getting it wrong – feeding a forecast where a measurement was assumed – is a category of bug that produces perfectly smooth wrong answers.

`seed` **is explicit and never defaults to the clock** – criterion 3.

`provenance` **is returned, not logged** – criterion 4. A caller that stores the outputs and discards the provenance has stored numbers it cannot defend.

7.10.2 The output is Chapter 5's tuple

The run does not return an array. It returns a stream of Chapter 5's tuples, in the same shape as a reading, with the fields read as Table 7.4 gives them.

Table 7.4: Chapter 5's tuple, read for a simulated value. Simulated data lands in the same store, in the same shape, as measured data; the binding is what tells them apart, and nothing downstream has to be rewritten to accept it.

Field	For a sensor reading	For a simulated value
value	what the sensor reported	what the model computed
unit	the sensor's unit	the model's unit, which must be the same one
`as_of`	when the world was in this condition	the **simulated** instant, not the wall-clock instant of the run

Field	For a sensor reading	For a simulated value
binding	which sensor on which asset	which **model version** on which asset, plus the run id
quality	the sensor's own health flags	whether the step converged, whether a surrogate refused, whether the value is inside the model's validated range
provenance	the acquisition path	the run that produced it

Two of those rows are where mistakes get made. The `as_of` of a simulated value is the instant in the *simulated* world, which for a prediction is in the future – and a store that silently stamps it with the wall-clock time of the run has turned a forecast into a measurement. And `quality` for a simulated value is not a sensor concept: it is the solver's convergence flag, the surrogate's refusal, the ensemble's spread. Give it a vocabulary in the twin's own terms rather than reusing the sensor's.

7.10.3 Five run modes

The same simulator, the same model, and five different jobs. What distinguishes them is where the initial state comes from, where the inputs come from, and what is done with the output – Figure 7.7 wires all five up at once, and Table 7.5 is the table to keep.

Table 7.5: The five run modes. The mode is an argument to the call because it fixes three things the simulator itself cannot know.

Run mode	Initial state from	Inputs from	Output goes to	Typical *RTF* needed
replay	an archived state at a past instant	the archive, measured	comparison against what happened	above 1, and the higher the better
shadow	the state block, now	the live data stream	the residual, against the live reading	about 1, paced
predict	the state block, now	a forecast, or a persistence assumption	the prediction store, with a horizon	well above 1
what-if	a **fork** of the state block	assumed, per scenario	a comparison across scenarios	above 1 times the number of scenarios
generate training data	sampled across the operating envelope	sampled across the operating envelope	a labelled data set	irrelevant; it is a batch job

Replay re-runs a past interval from an archived state against the inputs that really occurred, and compares the model's trajectory with the asset's. It is the cheapest verification a twin has and the first thing to build.

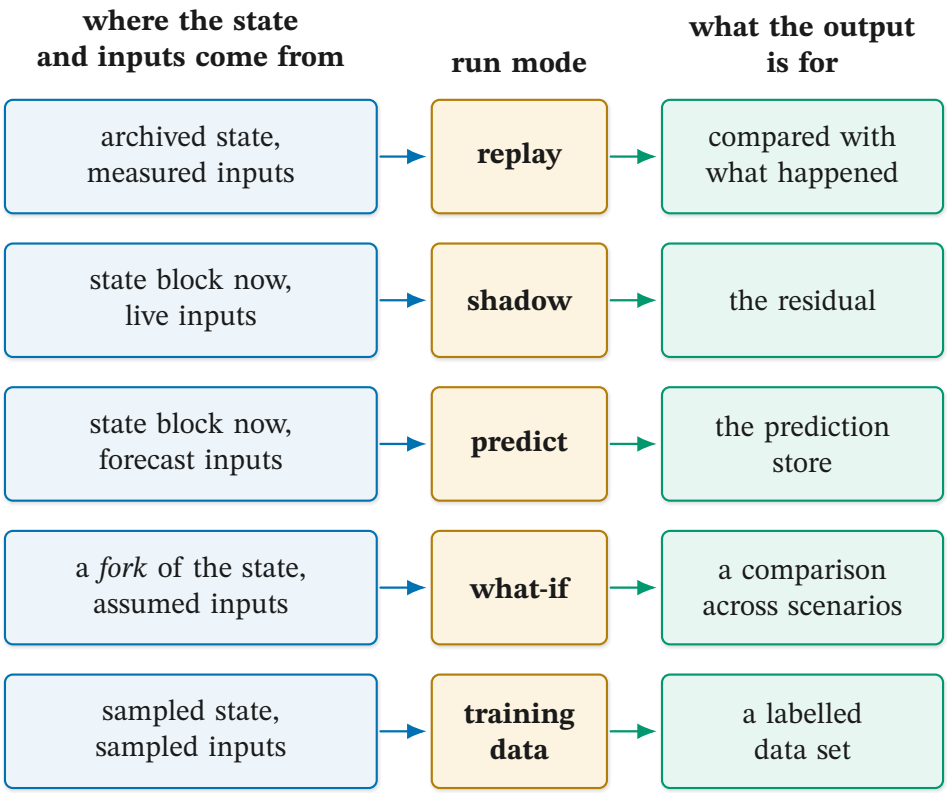


Figure 7.7: The five run modes. Each is the same call with a different source for the initial state and the inputs, and a different destination for the output: replay from the archive, shadow from the live state and live inputs, predict from the live state and a forecast, what-if from a forked state and an assumed input, and training-data generation from sampled states and sampled inputs.

Shadow is the mode Chapter 3's shadow twin runs continuously: the model steps alongside the asset from the same state, and the difference between its output and the incoming reading is the residual Chapter 2 already watches.

Predict runs from now into the future, with the inputs necessarily assumed – a wind forecast, a persistence assumption, a planned schedule. The honesty of a prediction is bounded by the honesty of its input series, and the horizon is bounded by the belief-decay argument of the Time section rather than by the budget.

What-if is predict, forked. The state block's belief is copied, several copies are run under different assumed inputs or different decisions, and the *differences* between them are the answer. The literature's definition is compact: a what-if simulation inspects the behaviour of the physical twin under given hypotheses called scenarios [187]. Two engineering points follow. The fork must be a real copy, or scenario B starts from the state scenario A left. And the comparison must use common random numbers, from the Stochastic Simulation section, or the difference between scenarios drowns in the difference between their weather. A published production twin runs this mode as an inner loop, generating candidate schedules, evaluating each with the discrete-event model and feeding the winner back to the operator or the execution system [183].

Generating training data is the mode that is not about this asset at all. Run the model across the operating envelope, record inputs and outputs, and use the result to train something – a surrogate, as the previous section described, or a controller, or a perception model. The survey of AI simulation by twins describes precisely this: virtual training environments in which agents are developed safely and efficiently on simulated, synthetic data, offered as the response to insufficient data volume and quality, with dataset generation the offline end of the practice

and training inside the environment the online end [111]. For this book, the point is that it is a **batch** job with no real-time requirement and a very large total compute, which is a different operational shape from the other four and should not share their infrastructure.

7.10.4 Where the simulator runs

Chapter 5 already decided this, and this section reads it off rather than re-deciding. That chapter's hot, warm and cold paths map onto the four placements of Table 7.6.

Table 7.6: Where a simulator runs, read off Chapter 5's path table. The row is chosen by the *RTF* the decision needs and the data the model must reach, in that order.

Placement	Chapter 5's path	Suits	Cost shape
embedded, on the controller	hot	a step function of a few lines, at the control rate	fixed, tiny, and capped by the hardware you already bought
edge, beside the asset	hot and warm	the lumped model, the mode machine, the reduced model	fixed hardware, no per-run charge, limited headroom for an ensemble
container, in a cluster	warm	ensembles, what-if fans, replay	per-run, elastic, the natural home of a bursty workload
cloud batch	cold	the field model, the surrogate campaign, training-data generation	per-core-hour, scheduled, and the one placement where a job queue is part of the design

Packaging is what makes a row portable. The **executable digital twin** is the literature's name for the artefact: a self-contained executable behaviour of an asset, usable by anyone at any point in the lifecycle, fast enough for its purpose – usually through model order reduction – with the packaged model carrying its own execution engine, and an explicit link back to the parent twin that keeps the two consistent and traceable [20]. Platforms take the same idea further down the stack, using containerisation, loose coupling and micro-services to compose twins dynamically and naming reproducibility as one of the properties that buys [188]. For a software engineer the reading is familiar: the simulator is an artefact with a version, a digest and a manifest, and it is deployed, not installed.

7.10.5 Run provenance

The last of the four criteria, written out. A run records:

- **the model**, by identifier and version, and the digest of the model file if it has one;
- **the solver**, by name, with its step size or tolerance and every option that was not the default;
- **the initial state**, or its digest, with the `as_of` it carried;

- **the inputs**, by reference to the series and its version, with a note of whether they were measured, forecast or assumed;
- **the seed**, and the stream assignment if the run was parallel;
- **the environment**, as a container digest or the equivalent [184];
- **the cost**: wall-clock time, steps taken, rate evaluations, and the achieved *RTF*.

That is six identifiers and a cost, and it is small enough to attach to every tuple the run produces. The test of whether it is complete is simple and worth running once: hand the provenance record to a colleague and see whether they can reproduce the numbers. If they cannot, the missing field is the one to add.

7.11 Choosing a Simulation Technique

Chapter 6 closed with a table of five model types against four criteria and four questions to walk it with. This section is the same shape for execution, and the two are meant to be read as a pair: that table chooses what to write, this one chooses what will run it and where.

7.11.1 The table

Table 7.7: The five model types against the technique each forces, what it can achieve, where it goes and who is needed. The third row's *RTF* is the reason the fourth row exists.

Model type	Technique it forces	Achievable *RTF*	Where it runs	Expertise needed
lumped first-principles	a solver; implicit or adaptive if stiff	10^5 to 10^7	embedded or edge	the domain's physics, plus a step size defended
discrete-state and discrete-event	an event scheduler, and replications	10^6 to 10^8 per run, times N	edge or container	the operation's rules, plus output analysis
high-fidelity field	a batch field solver; meshing, sparse solve, eigen-analysis	about 10^{-2}	cloud batch, off the critical path	a specialist package and a specialist
reduced-order and surrogate	a cheap evaluation online, an expensive campaign offline	10^3 to 10^9 online	edge online, cloud batch offline	the field model's, plus projection or fitting
probabilistic, Bayesian	an ensemble: N runs of whatever the wrapped type uses	the wrapped type's, divided by N	container, or edge if N and the step are small	probability, plus seed and stream discipline

Two entries in Table 7.7 are worth reading twice. The probabilistic row's *RTF* is the wrapped type's *divided by N*, which is the whole reason an ensemble over a lumped model is free and an

ensemble over a field model is impossible. And the reduced-order row has two placements and two rates, because it is the one type whose cost is paid somewhere other than where its benefit is taken. Choosing the formalism, and therefore the simulator, is a deliberate stage in a twinning system's design rather than something inherited from whatever tool was already installed [12].

7.11.2 Four questions

Walk the table with these, in order. Each removes rows.

1. **Rates, events, or both?** Quantities that change continuously want a solver; modes, arrivals and waiting want a scheduler. If the answer is both – and for any asset with operating modes it is – it is a hybrid model, and the extra question is how many switches an hour, because switches, not steps, set the cost.
2. **A number, or a spread – and can the step afford the runs?** A monitoring threshold takes a number. Anything with "when", "how long" or "how likely" in it takes a spread, which means N runs, which means the step function's cost is multiplied by N before it is compared with the budget. This is the question that most often sends a design back to a cheaper model type.
3. **Does the achievable *RTF* clear the decision's latency?** Take the latency from Chapter 1, the *RTF* from the table, and divide. If it clears, stop. If it does not, there are exactly four remedies and they are worth naming so that the choice is made rather than defaulted: shorten the horizon, coarsen the model, buy a surrogate with an offline campaign, or move the decision off the critical path.
4. **Where must it run, and does that placement allow the technique?** An embedded target may have no floating point worth the name, no room for an implicit solver's inner iteration and a hard real-time deadline that an adaptive solver's variable cost violates. An edge box can run the ensemble but not the campaign. Placement constrains technique, not only the other way round, and it is the question most often asked last and answered first.

The turbine's answers, returning to Chapter 6's five decisions. Power and thermal expectation: rates, a number, ten-minute latency, edge – an implicit or adaptive solver on two states, *RTF* around 10^7, and the only real decision was the step. Availability and repair planning: events, a spread, hours of latency, container – an event scheduler with a couple of thousand replications, and the arithmetic of the Discrete- Event section sizes them. Structural integrity: rates over a body, a number, seasonal latency, cloud batch – a field solver, and an eigen-analysis rather than a time-domain run. Load monitoring every second: the reduced tower at the edge, its basis taken from the eigen-analysis already paid for. Bearing replacement: a spread on a monthly latency, two thousand ensemble members over a four-operation step, anywhere at all. Five decisions, five techniques, four placements, one turbine.

7.11.3 Trusting a run, as arithmetic

Four checks make a run trustworthy *as arithmetic*. Each is cheap, each is mechanical, and a twin should run them on a schedule rather than when something looks wrong.

1. **Refine the step.** Halve h, or tighten the tolerance by an order of magnitude, and re-run. If the answer moves by more than your tolerance, the answer you had was the step size's. This is the mesh convergence study of the Field Solvers section, applied in time instead of in space, and it is the single most informative check in the list. Measure it the way the tolerance sweep of the Continuous- Time section was measured – reporting step counts, not only elapsed time, on a model small enough for the elapsed time to be noise [181].

2. **Swap the solver.** Run the same model with a different step kind – explicit against implicit, one adaptive family against another. Two independent methods agreeing is evidence about the arithmetic that neither gives alone; two disagreeing means one of them is wrong and you do not yet know which.
3. **Check a conserved quantity.** Energy, mass, charge, the number of parts in the system, the sum of the probabilities. Whatever the physics conserves, compute it at the start and the end and compare. This catches sign errors, unit errors and dropped terms that no amount of step refinement will, because a wrong model converges beautifully to the wrong answer.
4. **Record the seed and the versions.** Which is the provenance record of the previous section, and it is a check because it is what makes the other three repeatable next quarter.

And then stop. All four are **verification**: they establish that the equations were solved correctly. None of them establishes that the equations describe the turbine. That is **validation**, it is the credibility chapter's subject, and the distinction is load-bearing enough that standards bodies make it explicit – model credibility assessment, including verification, validation and uncertainty quantification, is to be applied throughout a twin's life cycle [87], and because a twin is dynamic, verification and validation are a continuous process performed periodically or on demand rather than a step completed once [48]. It is also a gap in practice rather than a solved problem: a recent survey observes both that ISO 23247 does not address verification, validation and uncertainty quantification at all, and that only about sixty per cent of the twin-driven AI simulation techniques it reviewed do anything to maintain the simulator's own fidelity over time [111].

7.11.4 For readers who know control theory

One analogy, gathered here rather than scattered through the chapter, for the readers who will find it useful; skip the paragraph if the words are unfamiliar, because nothing later depends on it.

A twin's simulator is doing the job the internal model does in a model-predictive controller: it is the plant model rolled forward over a horizon to produce a predicted trajectory that something else then acts on. The correspondences carry further than they usually do in analogies. The prediction horizon is bounded by model quality in both cases, which is the belief-decay argument of the Time section. The what-if fan is a candidate-input sweep. The real-time factor against the decision's latency is the constraint that decides how large a candidate set the controller can evaluate per interval, which is the classic implementation limit of the method. And the residual of Chapter 2 is the innovation. What is *not* shared is the closing of the loop: the controller commits its first move and re-plans, whereas a twin at the monitor, diagnose or predict patterns stops at the prediction and hands it to a person. The control chapter of a later part is where the difference is taken seriously; here it only explains why so much of this chapter's vocabulary will sound familiar to a control engineer, and why the state and estimator chapter, next, is the other half of the picture.

7.12 Simulating the Plant Demonstrator

The one section where the rig of Chapter 2 appears, with that chapter's symbols unchanged. It is here for a negative result: the rig is a perfectly good twin and it teaches almost nothing about choosing a simulator, and knowing *why* is worth a section.

7.12.1 The simulator is twelve lines

Chapter 3 named the rig's simulator without writing it. Chapter 6 showed that the rig's model is already a lumped first-principles model in sampled time,

$$m_{k+1} = m_k + g\,V_k - d\,T_c \tag{7.9}$$

Equation 7.9 is the whole model; here is the whole simulator around it.

```
def simulate(m0, doses, n_checks, theta):
    """Run the pot forward n_checks checks. theta = (g, d, T_c)."""
    g, d, T_c = theta
    m = m0
    trace = [(0, m)]
    for k in range(n_checks):
        V = doses[k]                  # millilitres at this check
        m = m + g * V - d * T_c           # the step: one line, no solver
        trace.append(((k + 1) * T_c, m))
    return trace
```

Twelve lines, and there is no solver in them. That is the section's first point and it is a general one: **the rig's step is already a difference equation in sampled time, so there is no solver choice to make.** The step size is not a numerical parameter at all – it is T_c, the check interval, fixed by the decision to look at the plant every six hours. Chapter 6 observed that this is Euler's step with $h = T_c$; the consequence for this chapter is that every question the Continuous-Time Simulation section spent pages on – explicit or implicit, what tolerance, what does stability cost – has already been answered by somebody choosing a check interval for reasons that had nothing to do with numerics.

7.12.2 Why no implicit solver

One state and one time constant. The moisture falls at d per hour and rises with a dose; there is no fast process hiding behind the slow one, so the system is not stiff, so there is nothing for an implicit step to buy. An implicit step here would solve a linear equation in one unknown to get the answer the explicit line already gave.

Worth being precise about the accuracy, because "it is not stiff" is not the same as "it is exact". Drying at d per hour over a six-hour interval is treated as $d\,T_c$ of loss, which assumes the rate is constant across the interval. If the drying rate in fact depends on how wet the soil is – it does, a little – then the six-hour explicit step is inaccurate in a way a shorter step would fix. The rig does not fix it, and it is right not to: the probe's own resolution is coarser than the error, so refining the step would improve a number nobody can measure. That is the honest form of "accurate enough", and it is a decision, not an oversight.

7.12.3 The watering schedule as an event queue

The rig has a second simulator, and Chapter 2 wrote it without naming it. The controller's day is a sequence of timed events: check at 06:00, check at 12:00, check at 18:00, and a pump that runs for a computed number of seconds when a check decides to dose. As an event queue that is three scheduled events a day, plus at most three pump-off events.

Six events a day, fifty steps of arithmetic between them. A whole growing season – one hundred and eighty days – is about a thousand events, which runs in a millisecond. The same season in a fixed-increment loop at one-second resolution would be 15.5 million steps of nothing. The Discrete-Event section's arithmetic, on a rig.

7.12.4 The rig's real-time factor

One step of the rig's model is a multiplication, a multiplication, an addition and a subtraction: call it four operations, plus the loop overhead, and call the whole step a microsecond on the rig's controller to be generous. One step covers $T_c = 6$ hours of simulated time, which is 21,600 seconds. So

$$RTF = 21600/10^{-6} = 2.2 \times 10^{10} \tag{7.10}$$

Twenty-two billion. Equation 7.10 means the rig's controller can simulate a full growing season in about a millisecond, which is why nothing in Chapter 2 ever discussed performance and why this chapter's central tension – can it run here, in time – does not arise at all. Every placement in the placements table is available. Every run mode is affordable. The choosing table's four questions all answer "anything".

7.12.5 Why no ensemble yet, and what would change that

The rig's twin reports a number: the predicted moisture, and the residual against the probe. It could report a spread instead – Chapter 6 gave it a posterior over the dose response g – and the ensemble to turn that posterior into a distribution of predicted moisture would cost a thousand runs of a four-operation step, which is nothing.

So why not? Because nothing downstream would use it. The decision is dose or do not dose, once every six hours, against a threshold; a spread around the predicted moisture changes that decision only when the threshold falls inside the spread, and a technician with a watering can is not going to act differently on "probably 22, could be 19 to 25" than on "22". Add the ensemble when the decision changes – when the rig grows a scarce water budget to allocate across pots, or a plant whose cost of over-watering is not symmetric with the cost of under-watering. Then the spread is what the allocation is computed from, and the ensemble earns its place.

Which is the honest conclusion of the section. The rig is the right scale for learning what a twin *is*; the turbine is the right scale for learning what a simulator *costs*. This chapter needed both.

7.13 Faded Examples

Three examples on Chapter 6's other systems, decreasing in how much is worked. None uses the rig.

7.13.1 Example 1, worked in full: the battery cell, stepped and then run many times

Continuous-time, then stochastic. Chapter 6's lithium-ion cell in a bus: an equivalent circuit with two states, the state of charge SoC and the polarisation voltage V_1, with $R_1 = 0.015$ ohms, $C_1 = 2000$ farads, a capacity $Q = 10800$ coulombs and a discharge current $I = 3.0$ amperes. Model-based battery prognostics work on exactly these equivalent-circuit models of cell voltage, or on physics-based degradation models [177].

Step 1: is it stiff? Two time constants. The polarisation pair settles with $R_1 C_1 = 30$ seconds. The state of charge falls at $I/Q = 2.78 \times 10^{-4}$ per second, so from full it takes $Q/I = 3600$ seconds to empty. The ratio is 120 to 1 – **mildly stiff**, which is the ordinary case and exactly the situation the appendix describes: a fast process finished almost immediately, sitting inside a slow one that is the reason the model exists.

Step 2: choose the step. Explicit stability needs h below about twice the fastest time constant, so $h < 60$ seconds. Accuracy on the polarisation transient needs rather better – a tenth of its time constant, $h = 3$ seconds, is the usual rule of thumb, and gives a voltage sag curve accurate to well under a millivolt. At $h = 3$ seconds an hour of discharge is 1,200 steps of about ten operations: twelve thousand operations, microseconds. So the answer is $h = 3$ seconds explicit, and the stiffness cost nothing at all.

Notice what made that painless: the *fast* state is the one the output depends on, so accuracy and stability pull the same way. Compare the turbine, where the fast state was the one nobody was asking about and the step it forced was pure overhead. That contrast is the whole of stiffness as an engineering matter.

Step 3: the ensemble, for capacity fade. The decision is not about this hour but about this cell's life: when will its capacity fall to eighty per cent of new, at which point the pack is retired? Chapter 6 gave the empirical fade law

$$Q(n) = Q_0 \left(1 - a\, n^b\right) \tag{7.11}$$

with n the cycle count. Take $b = 0.5$ – fade proportional to the square root of cycles, the common empirical form – and a fade coefficient a that is *not* known for this cell: from the fleet, it is normally distributed with mean 0.004 and standard deviation 0.0005. End of life is $Q/Q_0 = 0.80$, which by Equation 7.11 happens at

$$n_{\text{eol}} = (0.2/a)^2 \tag{7.12}$$

At the mean a, Equation 7.12 gives $(0.2/0.004)^2 = 2500$ cycles. But the answer wanted is not that number. Draw a from its distribution N times, compute n_{eol} for each, and the histogram is the answer: at $a = 0.0035$ the cell lasts 3,265 cycles, at $a = 0.0045$ it lasts 1,975, so a one-standard-deviation swing in a parameter nobody measured moves the retirement date by thirteen hundred cycles. Reporting "2,500 cycles" would have thrown that away. Algebraic reduced-order capacity models of exactly this form, paired with sampling methods for the parameter uncertainty, are reported as capturing model-form and parameter uncertainty in a way a plain neural-network ensemble does not [163].

Step 4: how many members? The spread of n_{eol} across the draws comes out at about 550 cycles. For the mean to be known to plus or minus 25 cycles at ninety-five per cent, $N = (1.96 \times 550/25)^2 = 1860$, so 1,900 members. Each member is one evaluation of Equation 7.12 – not a simulation at all – so the ensemble costs microseconds. If instead the decision needs the fifth percentile – "retire early enough that only five per cent of cells fail first" – size the ensemble for the percentile, which will need several times as many members, and say in the twin's output which statistic was asked for.

Step 5: the retraining trigger. The fade coefficient is a parameter with a posterior, and every month's capacity measurement updates it by Chapter 6's Bayes' rule. That is the twin working as designed. The trigger for something larger is a *systematic* drift – the posterior walking steadily in one direction over many months, which says the fade *law* is wrong for this duty cycle and not merely its coefficient unknown. A published framework does the same job the other way round, fitting an empirical capacity fade model and training a machine-learning model to predict its parameters from early-life features, reporting under two per cent error in distribution and under four per cent out of it [180] – and that out-of-distribution number is the validity check of the Making an Expensive Simulator Cheap section, in a battery's clothes.

7.13.2 Example 2, one step left to you: the 3D printer as a hybrid simulation

Hybrid. Chapter 6's desktop 3D printer, whose modes have been published as a state machine: StandBy, Connected on a device connecting, Operational on receiving a print file with preheating

before printing, pausing and cancelling while printing, back to StandBy when the job finishes, Error reachable from anywhere and a reset back to StandBy [178]. That machine is the discrete half. The continuous half is one rate rule per mode: the hot end's temperature, rising towards its set point under the heater's power and falling by loss to the chamber, with the extruder's filament consumption running alongside it while printing.

The hybrid structure is the interesting part. **Preheating** ends at a guard on a continuous state – the hot-end temperature reaching the material's print temperature – which is a zero-crossing to be located, not a flag to be polled. **Printing to paused** and **paused to printing** are guards on inputs, so no search is needed, but each one changes the heater's duty and therefore the rate: a solver restart each time. **Error** can fire from any mode and applies a state reset – the heater off, the extruder retracted – which is the expensive kind.

Your step. A four-hour print job on a printer that is paused twice by the operator, hits one filament-jam error and is resumed, and whose chamber-temperature controller cycles a cooling fan on and off with a guard on the chamber temperature every ninety seconds:

1. Count the solver restarts over the job, and separate them into those that need a zero-crossing search and those that do not.
2. The fan guard is the one to worry about. If the fan's threshold has no hysteresis and the chamber temperature is noisy around it, describe what the simulator does. Then say what a two-degree hysteresis band changes, in restarts per hour.
3. At an adaptive solver cruising at a one-second step and costing fifteen steps to climb back after a restart, what fraction of the four-hour job is restart overhead, with and without the band?

Hint. Work in restarts per hour before working in steps. The two pauses and the error are three restarts in four hours and are not the problem; the fan at ninety seconds is forty an hour, and a chattering fan is unbounded.

7.13.3 Example 3, posed as a problem: the industrial oven with a conveyor

Field, then reduced. Chapter 6's industrial oven with a conveyor, whose published relative is the induction furnace twin: steel bars passing through a furnace divided into five zones of induction heaters of four coils each, with pyrometers between the coils, entering a holding mode with reduced coil power and the bars oscillating slowly when something downstream stops [179]. A twin of it must answer, for each bar, whether its core reached temperature – a question about the inside of a body, which the pyrometer cannot see and only a field model can answer. Two small published cases, a line-following robot and an incubator, are offered in the literature as the tractable stand-ins for an automated guided vehicle and an industrial convection oven respectively, which is the right scale to prototype this at [189].

The problem. The oven runs a bar through in eight minutes. The field model of one bar is a three-dimensional transient thermal model with about 200,000 elements, and a time-domain run of one bar's passage takes forty minutes on sixteen cores.

1. **Size the field run.** Compute the real-time factor of the field model as stated, and say which of the placements table's four rows it belongs in. Then say what would have to change for it to move up a row, and whether any of those changes is available.
2. **Cost a mesh convergence study.** The quantity of interest is the core temperature at the exit. Design a study over three mesh refinements, say what you would compare and what tolerance would let you stop, and give the total core-hours.
3. **Design the campaign.** A reduced or surrogate model has to run at belt speed – one bar every eight minutes, with an answer before the bar leaves the oven. The inputs are the

bar's diameter, its entry temperature, the five zones' power settings and the belt speed: eight inputs. Say why a full factorial is not available here, choose a design, and give its cost in core-hours against the forty-minute run time.

4. **Choose the run mode.** Which of the five run modes does the twin use this surrogate in during normal production, and which does it use during the holding mode the source describes? They are not the same one, and the difference is where the inputs come from.
5. **Name what is still missing.** The surrogate answers for a bar whose inputs lie inside the campaign. A bar of a new alloy arrives. Say what the validity check does, what the twin reports, and what the retraining trigger costs in days.

7.14 Summary

Six items, returning to the six objectives.

1. **A simulator is an engine, a step, a clock and a contract** (objective 1). The engine owns the loop and the clock; the step is either a solver over rate rules or a scheduler over an event queue, and hybrid models need both; the clock relates simulated time to wall-clock time and their ratio is the real-time factor; the contract is Chapter 4's interface written out. Four criteria judge a simulator: inside the decision's time budget, restartable from a state the twin supplies, deterministic enough to compare with itself, and able to say what it ran. The last two are software criteria and are the two most often missing.
2. **The step size is an engineering choice, and stiffness is what forces it** (objective 2). Explicit, Heun, Runge-Kutta, implicit and adaptive, priced in rate evaluations; closing the turbine's torque loop puts a 3.5-second time constant in the same model as a 2000-second one, and an explicit solver must then step at about one second – six hundred steps per SCADA reading – or diverge in two steps. An implicit step at the full six hundred seconds costs one small equation solve and lands on the answer. Algebraic constraints are solved after each step; algebraic loops need an inner iteration nobody budgeted for; and a twin re-initialises every cycle, so initialisation is a per-cycle cost.
3. **A discrete model needs a queue, and one run of a stochastic model answers nothing** (objective 3, objective 4). Next-event advance turns 31.5 million empty seconds into a few thousand events. Then the real work: seeds, independent streams, replication, warm-up, run length, and a confidence interval reported rather than a mean. The vessel's twenty and a half hours is one sample; two thousand two hundred replications pin the mean to half an hour, and ten times the precision costs a hundred times the runs. Hybrid models cost switches, not steps: a zero-crossing search to locate the instant, a solver restart to pay for the reset, and a hysteresis band to stop chattering.
4. **An ensemble is a workload, and its size is arithmetic** (objective 4). Draw the parameter from its posterior once per member, step to the threshold, and the histogram of crossing times is the remaining-life distribution – Chapter 6's three-centre mixture with $N = 3$ and the noise switched off. Common random numbers are what make a comparison affordable; the seed and the container digest are what make a run repeatable; and a tail percentile needs many more members than a mean.
5. **The real-time factor decides where a simulator lives, and the belief decides how far ahead it runs** (objective 5). Below one, batch only – which rules the field model out of the loop entirely, at $RTF = 0.01$, and sends its answer to a reduced model bought with an offline campaign whose grid of five inputs at five levels is 3,125 runs. Around one, pace it. Well above one, divide by the decision's latency to see how many futures fit. And stop predicting at the horizon where the belief has leaked away, which is a property of the system and not of the simulator.

6. **A run is a call that returns numbers and a provenance record** (objective 6). Initial state in, input series in, horizon in, mode in, seed in; simulated-data tuples out, with the binding naming a model version and the `as_of` naming the simulated instant. Five run modes – replay, shadow, predict, what-if, generate training data – distinguished by where the state and inputs come from. Four numerical checks make a run trustworthy as arithmetic: refine the step, swap the solver, check a conserved quantity, record the seed and the versions. None of the four says the answer is right about the asset.

The sentence to carry forward: **this chapter produces the twin's expectation; the estimator, next, is how the twin corrects that expectation with what the sensors actually said.**

7.15 Exercises

Solutions or hints follow the exercises; each says which. The objective each exercise serves and its difficulty are given in brackets. None of the exercises uses the plant demonstrator.

Exercise 1 (Objective 2, easy). A thermal model of a transformer has two states: the winding, with a time constant of 90 seconds, and the oil tank, with a time constant of 4 hours. The twin needs the oil temperature once a minute.

(a) Is the model stiff? Give the ratio. (b) What is the largest step an explicit solver can take? (c) How many steps per minute of simulated time is that, and how many of them are there for the oil's sake?
(b) Name the two remedies and say what each costs.

Exercise 2 (Objective 2, medium). You are handed a solver with a relative tolerance setting and a model you did not write. Design the experiment that tells you what tolerance to use, and say what you will report. You have one afternoon.

Exercise 3 (Objective 4, easy). A discrete-event model of a spare parts depot gives, over 400 replications, a mean stockout duration of 11.2 hours with a sample standard deviation of 9.0 hours.

(a) Give the ninety-five per cent interval on the mean. (b) How many replications are needed to pin it to plus or minus one hour? (c) The planner actually wants the ninetieth percentile. Say why (b) is the wrong calculation and what you would do instead.

Exercise 4 (Objective 4, medium). Two maintenance policies are compared by simulation: 500 replications each, seeds drawn independently. Policy B looks 0.4 hours better, with a standard error on the difference of 0.5 hours – not significant. Your colleague proposes running 5,000 replications of each. Propose something cheaper that would probably settle it, say precisely what it requires of the simulator, and say what would make it fail.

Exercise 5 (Objective 3, medium). A pump model has a guard $p \geq 4.0$ bar that switches it from idle to running, and $p < 4.0$ that switches it back. The pressure signal has noise of about 0.1 bar.

(a) Describe what the simulator does when the pressure sits at 4.0.
(b) Give a hysteresis band that fixes it, and say how you chose the two thresholds. (c) Name one alternative remedy and one case where the band is the wrong answer.

Exercise 6 (Objective 2, objective 3, medium). An adaptive solver runs a hybrid model at a cruising step of 0.5 seconds and needs 12 steps to return to cruising after a restart. The model switches mode 20 times an hour, and each switch needs a zero-crossing search costing 6 extra rate evaluations.

(a) What fraction of an hour's rate evaluations is switch overhead?
(b) The model is changed so that it switches 2,000 times an hour. Recompute. (c) At what switch rate does the solver spend half its effort on switching, and what would you do about it?

Exercise 7 (Objective 5, medium). A reduced model covers 1 second of simulated time in 1.2 milliseconds of wall-clock time on the edge box it is deployed to.

(a) Give its real-time factor. (b) The decision it serves has a 30-second latency budget. How many 10-minute what-if scenarios fit?
(b) The model is moved to a container in a cluster where it runs three times faster but the round trip to fetch the state costs 250 milliseconds. Recompute (b) and say which placement you would choose.

Exercise 8 (Objective 6, open-ended). You inherit a twin whose simulator returns arrays of numbers and writes a line to a log file per run. Write the provenance record you would add, field by field, and for each field say what question it answers and what goes wrong without it. Then say how you would test that the record is complete.

7.15.1 Solutions and hints

Exercise 1. (a) 4 hours is 14,400 seconds, so the ratio is $14400/90 = 160$: stiff. (b) About twice the fastest time constant, so roughly 180 seconds – but accuracy on the winding wants perhaps 9 seconds, a tenth of its constant, and whichever is smaller governs, so 9 seconds. (c) Between 7 steps a minute (at the stability bound) and about 7 steps per minute at the accuracy bound – at $h = 9$ seconds, 6.7 steps a minute, essentially all of them there for the winding, since the oil moves by about a thousandth of its range in a minute. (d) An implicit step, which costs an equation solve per step and removes the stability bound but not the accuracy one; or an adaptive solver, which costs a variable per-step budget – awkward on a hard real-time target – and spends its effort where the model needs it.

Exercise 2. Hint. Sweep the tolerance over several orders of magnitude, as the tolerance-sweep study in the Continuous-Time section did, and at each setting record both the quantity of interest and the **number of steps**. Plot the quantity against the tolerance: it will move and then stop moving, and you want the loosest tolerance in the flat region, with a safety margin. Report the step count rather than the wall-clock time if the model is small enough for the timing to be noise, and report both if it is not. Then repeat once with a different solver, because a tolerance that is fine for one is not automatically fine for another.

Exercise 3. (a) $1.96 \times 9.0/\sqrt{400} = 0.88$ hours: 11.2 plus or minus 0.88, so about 10.3 to 12.1. (b) $N = (1.96 \times 9.0/1.0)^2 = 311$ – fewer than you already have, so the interval is already inside one hour. (c) The formula is the interval on the **mean**. A ninetieth percentile is estimated from the upper tail of the sample, so its precision depends on how many replications landed out there – roughly forty of the four hundred – and it needs several times as many replications for comparable precision. Estimate the percentile directly from the order statistics, put a bootstrap interval on it, and size N from that interval rather than from the mean's.

Exercise 4. Re-run both policies with **common random numbers**: the same seeds, paired member for member, so the weather, the arrivals and the failures are identical under both policies and the difference is the policy's. The requirement is independent random streams – policy B must draw its weather from the same stream at the same points as policy A, which means the streams must be indexed by what they are for and not consumed in call order. It fails when the two policies do not consume the same random draws in the same order – for instance if B dispatches a second vessel and thereby asks the weather model a different number of questions

– in which case the pairing breaks and you are back to independent runs, or to restructuring the model so that every random draw is requested from a named stream at a fixed point.

Exercise 5. (a) It chatters: the guard fires, the transition changes the state, the reverse guard immediately fires, and the solver truncates step after step while simulated time barely advances – wall-clock time passing with the simulated clock stationary. (b) Switch on at 4.1 bar and off at 3.9; the band must be comfortably wider than the noise, so about two standard deviations either side, and it should also be narrower than any pressure change the twin must detect. (c) A minimum dwell time in each mode, or filtering the guard's input. The band is the wrong answer when the real asset genuinely does switch at a single threshold and the rapid switching is a real behaviour the twin exists to report – a relay welding shut, for instance – in which case the fix is to model the switching, not to suppress it.

Exercise 6. (a) Cruising at 0.5 seconds is 7,200 steps an hour. 20 switches cost $20 \times 6 = 120$ extra rate evaluations for the searches and $20 \times 12 = 240$ steps of climb-back: about 360 against 7,200, or five per cent. (b) 2,000 switches cost 12,000 search evaluations and 24,000 climb-back steps against 7,200 cruising steps – the solver now does five times as much switching work as useful work, and in practice never reaches its cruising step at all. (c) Setting climb-back plus search equal to the cruising steps gives roughly $18\, n_{\text{switch}} = 7200$, so about 400 switches an hour, one every nine seconds. Above that, either the mode structure is too fine-grained for the question and should be coarsened, or the switching is chattering and needs a band, or the model should be reformulated so that the fast switching becomes a continuous approximation.

Exercise 7. (a) $RTF = 1/0.0012 = 833$. (b) 30 seconds of budget covers $833 \times 30 = 25{,}000$ seconds of simulated time, which is about seven hours, or 41 ten-minute scenarios. (c) Three times faster is $RTF = 2500$, but 250 milliseconds of the 30-second budget is gone to the round trip, leaving 29.75 seconds: $2500 \times 29.75 = 74{,}375$ seconds, about 124 scenarios. The cluster wins on throughput by a factor of three. Choose it if the what-if fan is the workload and the round trip is paid once per fan; choose the edge if the same simulator also has to run the shadow mode against a live stream, because there the 250 milliseconds is paid every cycle and the availability of the link becomes a dependency the asset does not otherwise have.

Exercise 8. Hint. The seven fields of the Run Provenance subsection are the answer's skeleton: model and version, solver and settings, initial state and its `as_of`, inputs and whether they were measured, forecast or assumed, seed and stream assignment, environment digest, and cost. For each, the question it answers is "what would change the numbers if it changed" – which is also the test for whether a field is missing. The completeness test is the one in that subsection: hand the record to a colleague with access to the same systems and ask them to reproduce the run. Whatever they have to come back and ask you for is the field you left out. Do it once per twin, not once per run.

7.16 Further Reading

Grouped by technique, one line per entry saying what you would go to it for, and saying where a worked example is the book's own construction rather than a source's.

- *What a simulator is:* [20] for the executable digital twin – a self-contained, deployable, fast-enough model carrying its own execution engine, which is the clearest published statement of what a simulator becomes inside a twin; [186] for high-fidelity simulators and reduced-order modelling named as twin enablers, with the challenge-by-technology table worth a read on its own. The four parts and the four criteria are the book's own framing.
- *Continuous-time execution:* [166] at book length for equation-based modelling, causality assignment, the implicit routine that algebraic loops force, and the Runge-Kutta family;

[181] for a measured tolerance sweep and the plain statement that events stop a variable-step solver; [169] for adaptive solvers as the customary answer to stiffness and for what moving a coupling boundary onto the slow variable buys; [167] for equation-based models reaching a running system as compiled artefacts. The turbine's three-ways table and its closed-loop time constant are the book's own.

- *Discrete-event execution:* [166] again for DEVS stated as a formalism, with worked code; [182] for what it takes to turn an existing discrete-event model into a twin, as a manufacturing case study; [183] for a published twin that calls its discrete-event model in an optimisation loop; [12] for formalism choice as a design stage. The vessel ensemble and its replication arithmetic are the book's own.
- *Hybrid execution:* [91] for hybrid automata of machine and process states in manufacturing twins, and for the plain finding that fidelity is paid for in simulation time; [181] and [169] again, for the two halves of what a switch costs – the stopped integration and the search for the instant.
- *Stochastic execution and reproducibility:* [11] for the sampling methods a continuous posterior needs and why they are expensive; [174] for a surrogate built specifically to survive a sampler; [185] for a twin whose output is a family of trajectories and a risk-aware decision taken from them; [184] for computational reproducibility and what a container actually pins; [31] for reproducibility and replicability defined for twin research, and why so little of it is either; [188] for a platform that treats reproducibility as an architectural property.
- *Field solvers:* [40] for the honest statement that a run takes hours or days and that a study needs many of them; [172] and [165] for finite-element model updating from measured natural frequencies, which is the eigenvalue run this chapter says is the field model's most useful one; [171] for a calibrated model computing modes no experiment could reach.
- *Buying the time back:* [66] for a component-based library of reduced-order models and the many-query argument that motivates it; [92] for the definition of a surrogate as a mimic of a model rather than of an asset; [174] for the multi-fidelity trick; [173] for reduced models argued for as reusable assets; [90] for fit-for-purpose as a principle and for the cost of training data named as an open gap. The campaign arithmetic and the validity-check listing are the book's own.
- *Time:* [141] for measured time discrepancy, measured catch-up, and the finding that a threshold set from ideal conditions is exceeded in normal ones – the most useful single paper for this chapter's Time section; [91] for the twinning rate as distinct from the sampling rate; [164] for a turbine twin built to run in real time beside its asset.
- *Driving a simulator, and run modes:* [187] for a compact definition of what-if simulation; [111] for the training-data run mode surveyed in full, and for the observation that most surveyed techniques do nothing to maintain the simulator's own fidelity; [183] again for what-if as an inner loop.
- *Trusting a run:* [87] for where a standard puts model credibility assessment; [48] for verification and validation as continuous rather than as a step; [90] for verification and validation across the whole twin ecosystem. The four numerical checks are the book's own list.
- *The faded examples:* [177] and [163] for equivalent-circuit and algebraic capacity models of a cell; [180] for a fade model whose parameters are predicted from early life; [178] for the printer's published state machine; [179] for the induction furnace and its holding mode; [189] for two small, fully worked cases at a scale you can rebuild.
- *Deliberately absent:* **co-simulation and distributed simulation**. Running two or more simulators together is a large literature with its own standards, and both the literature and the standards belong to the platforms and standards chapter of Part III rather than here. If you have come to this chapter looking for the Functional Mock-up Interface or the High

Level Architecture, that is where they are.

Part III

The Ecosystem

Part IV

Appendix

Appendix A

The Mathematics You Need

A.1 Before You Start

Who this is for. A software engineer who took one calculus course and one probability course some years ago, has used neither since, and wants the minimum this book leans on. You write loops, records and tables for a living. Everything in this appendix is built from one of those three.

What it is not. A mathematics course. Nothing here is proved. Where a result is stated – that a smaller step gives a smaller error, that a sum of many small independent errors is bell-shaped – it is shown on numbers you can check with a calculator, and the Further Reading section says where the proof lives. If you want to know *why* a rule holds rather than *what* it does, this is the wrong document, and it says so rather than pretending otherwise.

How to use it. Each chapter's Before You Start names the sections of this appendix it assumes. Read those and skip the rest. The Where Each Section Is Used section at the end is the same map read backwards: from a section here to the chapters that lean on it, so that a reader who arrives from one pointer can see what else the same idea unlocks. One worked instance – a cup of coffee cooling on a desk – runs through every section, so a reader who has followed one section already knows the example the next one starts from. The body of the appendix names no chapter; the closing table does that once, for all of them.

One convention, stated once. Every displayed equation in this book is one line. A rule that needs two relations writes them on one line separated by a comma; a matrix is written row by row with semicolons between the rows. This is the book's mapped-mathematics convention, and it is what lets the same equation read as plain text in the Markdown source and as typeset mathematics in the printed page. Symbols are written the way you would type them: $*$ is multiplication, / is division, $\hat{}$ is a power, _ is a subscript, and dT/dt is a derivative. The Notation section collects all of it in one table.

Pseudocode. Wherever a piece of mathematics is something a program does, it is also written as a short listing in an invented, Python-like pseudocode: a loop, a record, two nested loops, a priority queue. The listings are the mathematics restated for a reader who thinks in code; none is meant to run as written.

Learning objectives. By the end of this appendix you will be able to:

1. **Read** a differential equation aloud, **name** its state, its parameters and its initial condition, and **compute** the rate at a given state.
2. **Write** a differential equation in its integral form, **solve** the cooling equation in closed form, and **step** any equation forward by hand with Euler's method and one better method, **saying** what a smaller step buys and costs.
3. **Multiply** a matrix by a vector as two loops, **transpose** a matrix, **invert** a two-by-two, **write** a system of differential equations in state-space form, and **read** a mode and its nat-

ural frequency off a small system.

4. **Compute** a probability from counts, **apply** conditional probability and Bayes' rule with numbers, **test** two events for independence, and **read** a uniform and a Gaussian density for a probability, a mean and a standard deviation.
5. **Write** a Markov chain's transition matrix, **predict** a belief one step ahead, and **correct** it against a noisy reading.
6. **Compute** the cost of a simulation run and its real-time factor, and **decide** from that number whether a model can live inside a twin.

The exercises at the end name the objective each one serves.

A.2 Quantities, Rates and Equations

A.2.1 A quantity with a history

A temperature sensor on a cup of coffee reports a number every second. Written down, the numbers are a column in a table: a time and a value on each row. In code the value is a variable, T, and the table is its history. Mathematics writes the same thing as $T(t)$ – the temperature *at* time t – and drops the bracket, writing T, whenever the moment does not matter. Both spellings appear in this book; they name the same variable. A **quantity**, in this appendix, is a number that changes over time, and its history is the table.

Two other quantities appear in the same story. The room is at T_a, which does not change over the ten minutes we care about, and the coffee starts at $T(0) = 80$ degrees. A quantity that does not change over the time span you are modelling is a **constant** for that model, even if it is a variable in the world.

A.2.2 The rate: how fast it is changing now

Look at two neighbouring rows of the table. The coffee read 80.0 at minute zero and 74.3 at minute one. It lost 5.7 degrees in one minute, so its **rate of change** over that minute was about -5.7 degrees per minute. Take two rows closer together – a second apart – and the rate over that second is a little steeper, closer to -6.0, because the coffee cools fastest when it is hottest. The **derivative** is the rate you would read from two rows infinitely close together: how fast the quantity is changing *right now*, not on average over some interval. It is written dT/dt and read aloud as "d T d t" or, more usefully, "the rate of T". The units are the quantity's units per time unit: degrees per minute here.

Three things about the rate that a programmer may not have been told:

- **It is a quantity too.** dT/dt has a value at every moment and a history of its own. It can be plotted; it can be measured, roughly, as the difference of two rows divided by the time between them.
- **Its sign is direction.** Negative means falling, positive rising, zero means the quantity is, for that moment, not moving.
- **It is local.** The rate at minute three says nothing about minute four. What connects the two is an equation.

A.2.3 Ordinary Differential Equations

What they are for. Most of what an engineer knows about a physical system is not a formula for its future but a statement about how it changes: heat flows from hot to cold in proportion to the difference; water leaves a tank in proportion to the pressure at the outlet; a population grows

in proportion to its size. Each of those is a rule for the *rate* of a quantity in terms of the quantity's present value. An **Ordinary Differential Equation** (ODE) is exactly that: an equation in which the unknown is a quantity's whole history and the equation relates the quantity's rate to its present value and a few constants. "Ordinary" because there is one independent variable, time; "differential" because a derivative appears in it. The equation is the model; the history it implies is what a simulator computes, and what a twin compares against the sensor.

Newton's cooling law as the example. Newton observed that a warm object loses heat in proportion to how much warmer it is than its surroundings. As an equation for the rate:

$$\frac{dT}{dt} = -k\,(T - T_{\mathrm{a}}) \tag{A.1}$$

Read Equation A.1 aloud: "the rate of T is minus k times the excess of T over the room". Everything an ODE has is on show. The **state** is T, the one quantity whose history the equation fixes. The **parameters** are the constants in the rule: k, in units of one over minutes, which says how quickly this cup in this room loses heat, and T_{a}, the room temperature. Change the cup for a vacuum flask and k falls; move the cup outside and T_{a} falls. Neither is a quantity the equation moves; both are numbers the equation uses. When this book speaks of parameters collectively it writes θ.

An equation alone does not say where the coffee is. It says how fast it is moving *given where it is*. To fix an actual history you also need to know where it started: the **initial condition** $T(0) = 80$. The pair – equation plus initial condition – fixes the whole future history, the **trajectory**, and nothing less does. Two cups obeying the same equation from different starting temperatures follow different trajectories; two cups from the same starting temperature with different k do too.

Order, linearity, and the words you will meet. The cooling law is **first-order**, because only the first derivative appears; a vibrating structure obeys a **second-order** equation, in which the acceleration d^2x/dt^2 appears, and the Many Quantities section shows how a second-order equation is rewritten as two first-order ones. The cooling law is also **linear**: the rate is a weighted sum of the state and a constant, with no square, product or function of the state in it. A draining tank, whose outflow goes with the square root of the level, is **non-linear**. Linear equations are the ones with formulas for their solutions and the ones a matrix can hold; non-linear ones are the ordinary case in the world and are handled by the loop of the next two sections.

Difference equations. Every worked example in this appendix ends up computing the trajectory step by step, and the rule a step follows has its own name. A **difference equation** gives the next value of a quantity from its present value, indexed by step rather than by time:

$$T_{k+1} = F(T_k) \tag{A.2}$$

In Equation A.2 the subscript k counts steps, so T_k is the value after k of them and T_0 is the initial condition. A difference equation is the discrete cousin of a differential equation: the same idea of "given where you are, here is where you go", with the step of time built in rather than shrunk to nothing. Some systems are difference equations to begin with – a bank balance compounding monthly, a counter incremented per event, a sensor that reports once per interval – and a differential equation becomes one the moment a program steps it, which is the subject of the From an Equation to a Loop section. The pair appears everywhere in this book: a differential equation is how the physics is written down, and a difference equation is what the twin actually runs.

A.2.4 Worked example 1: reading the equation at three temperatures

Take $k = 0.1$ per minute and $T_{\mathrm{a}} = 20$ degrees. At $T = 80$, the excess over the room is 60 degrees, so the rate is $-0.1 \times 60 = -6$ degrees per minute. The reasoning behind each move: subtract

the room temperature first, because the equation is about the *excess* and not the temperature; then multiply by k, whose units of one over minutes turn degrees into degrees per minute; then attach the sign, which says the coffee is cooling.

At $T = 50$ the excess is 30 and the rate is $-0.1 \times 30 = -3$: half the excess, half the rate. That proportionality is the whole content of the equation.

At $T = 20$ the excess is zero and so is the rate. The coffee has reached the room and stops moving. A value at which the rate is zero is an **equilibrium**, and this equation has exactly one.

A.2.5 Worked example 2: the same equation, a different parameter

Keep the coffee at $T = 80$ and change the cup. A thin paper cup has $k = 0.2$; the rate at 80 becomes $-0.2 \times 60 = -12$ degrees per minute, twice as fast. Now keep the paper cup and carry it into a warmer room at $T_\mathrm{a} = 30$: the excess is 50 and the rate is $-0.2 \times 50 = -10$. Same state, same equation, different parameters, different rate. When a model is "kept current" by re-estimating its parameters, this is what moves: not the equation, and not the state, but k and T_a.

A.2.6 Worked example 3: a difference equation from a watering rig

A potted plant on a bench has a soil-moisture probe read every ten seconds, and between waterings the soil dries at a steady $d = 1.0$ percentage points per hour – a number read off two probe readings eight hours apart, $d = (44 - 36)/8 = 1.0$. As a differential equation the drying is the simplest kind, a constant rate:

$$\frac{dm}{dt} = -d$$

State m, the soil moisture; parameter d; initial condition $m(0) = 44$. Because the rate does not depend on m, the trajectory is a straight line – m falls one point every hour. As a difference equation stepped once per probe reading, with the reading period $T_\mathrm{r} = 10$ seconds written in hours:

$$m_{k+1} = m_k - d\,T_\mathrm{r}$$

Ten seconds is 1/360 of an hour, so each step lowers m by $1.0/360 = 0.0028$ points, and 360 steps later, one hour on, m has fallen by one point as it should. The rig's controller was running a difference equation all along; the differential equation is its statement of the physics, and the difference equation is what the code computes. Because the rate here does not depend on the state, the two agree exactly; the next two sections are about what happens when it does.

A.3 The Integration

A.3.1 From a rate to a total

A differential equation says how fast a quantity is moving. To know where the quantity *is*, you have to add up all that movement from the start until now. Adding up a rate over time is **integration** – the area under the rate's curve – and it is the second half of the calculus course you took. If the rate of T is $f(T)$, then the temperature at time t is the temperature at the start plus the accumulated rate:

$$T(t) = T(0) + \int_0^t f\big(T(s)\big)\,ds \tag{A.3}$$

Equation A.3 is the **integral form** of the differential equation, and it says the same thing the differential form says, read the other way round. The differential form asks "how fast, right now?"; the integral form asks "how much, so far?". The variable s inside the integral is a stand-in for every moment between zero and t, and ds is the width of the sliver of time each rate value applies to. Nothing has been solved yet: T appears on both sides, inside the integral and out. But the form is what every solver in the next section is approximating, and it is what a closed-form solution, where one exists, comes from.

A.3.2 Solving the cooling equation in closed form

For the cooling law the integral can be done by hand, because the equation is linear. The steps are the ones your calculus course drilled, and each is shown so the result can be checked rather than believed. Write the excess over the room as $y = T - T_a$; since T_a is constant, $dy/dt = dT/dt$, and the equation becomes $dy/dt = -k\,y$. Separate the two variables, putting everything in y on one side and everything in t on the other: $dy/y = -k dt$. Integrate both sides from the start to time t: the left side gives $\ln(y(t)) - \ln(y(0))$, the right gives $-k\,t$. Undo the logarithm with the exponential, and put T back:

$$T(t) = T_a + \big(T(0) - T_a\big)\, e^{-k\,t} \tag{A.4}$$

Equation A.4 is the **closed-form solution**: a formula that gives the temperature at any time directly, with no stepping. The excess over the room decays exponentially, and $1/k$ – ten minutes for the coffee – is the **time constant**, the time in which the excess falls to about 37 per cent of what it was. Every cooling curve drawn in this appendix, and every "exact" value the worked examples compare a loop against, comes from this line.

Why the loop is still needed. The trick above – separate the variables, integrate each side – works because the equation is linear and has one state. Add a second coupled quantity with a non-linear exchange between them, or a rate that depends on a measured input switching on and off, and there is no formula. For almost every equation a twin uses, the integral form is exact and unsolvable by hand, and the trajectory is produced by approximating the integral one sliver of time at a time. That is the loop, and it is why an equation with no closed form is no obstacle to a twin.

A.3.3 Worked example 1: the closed form at three minutes

Check the formula against the numbers the rest of the appendix uses: $k = 0.1$, $T_a = 20$, $T(0) = 80$. At $t = 1$, $\exp(-0.1) = 0.9048$, so $T(1) = 20 + 60 \times 0.9048 = 74.29$. At $t = 3$, $\exp(-0.3) = 0.7408$, so $T(3) = 20 + 60 \times 0.7408 = 64.45$. At $t = 10$, one time constant, $\exp(-1) = 0.3679$, so $T(10) = 20 + 60 \times 0.3679 = 42.07$: the excess of 60 has fallen to 22.07, which is 36.8 per cent of it, as the time constant promised.

A.3.4 Worked example 2: a question the closed form answers directly

When does the coffee reach 30 degrees? The loop would have to run until it got there; the formula is solved for t. Set $T(t) = 30$, so the excess is 10 out of an initial 60: $\exp(-0.1\,t) = 10/60$. Take logarithms: $-0.1\,t = \ln(1/6) = -1.792$, so $t = 17.9$ minutes. Questions of the form "when will it reach", "how long until", and "what is it at exactly this time" are where a closed form earns its keep, and they are the questions a monitoring dashboard asks.

A.3.5 Worked example 3: the integral form for the drying soil

The drying equation $\frac{dm}{dt} = -d$ has a rate that does not depend on the state, so its integral form can be integrated at once: $m(t) = m(0) + \int_0^t (-d)\,ds = m(0) - d\,t$. With $m(0) = 44$ and $d = 1.0$, $m(8) = 44 - 8 = 36$, which is the reading the drying rate was computed from in the first place. The lesson is the one the closed form of the cooling law taught from the other side: the integral form is always right, and the only question is whether the integral can be done by hand. Here it can, because the integrand is a constant; for the coffee it could, because the equation is linear; for the coupled coffee and mug of the Many Quantities section it still can, with more algebra; for a draining tank with a square root in it, it cannot, and the loop takes over.

A.4 From an Equation to a Loop

A.4.1 What a solver is

A **solver** is the program that turns a differential equation into a trajectory. It holds the state, chooses a step of time, uses the equation to move the state across that step, records the result, and repeats until the end of the run. Every solver is an approximation of the integral form of the previous section: the integral adds up the rate over every instant, and a solver adds it up over a finite number of slivers instead. The methods differ only in how they estimate the rate across one sliver. This section builds the simplest one by hand and then shows two better ones at the same level of detail.

A.4.2 Euler's step: the integral over one sliver

Take the integral form of the previous section over one short sliver of time, from t to $t + h$. The exact change in T is the area under the rate curve across that sliver. The crudest estimate of that area is a rectangle: the rate at the *start* of the sliver, held constant across its width h. Rate times width is the rectangle's area, and adding it to the starting value gives **Euler's step**:

$$T_{k+1} = T_k + h\,f(T_k) \tag{A.5}$$

New value equals old value plus rate times step. In Equation A.5, h is the **step size**, the time one step advances; $f(T_k)$ is the equation's right-hand side evaluated at the present value, $-k\,(T_k - T_a)$ for the coffee; and the subscript k counts steps. The step counter k and the cooling parameter k are different letters in the printed mathematics and the same letter in plain text; this appendix writes the parameter as k and the step in the subscript position only, and the Notation section says how the book keeps them apart. Euler's step is a difference equation – the previous section's $F(T_k)$ with $T_k + h\,f(T_k)$ filled in – which is the sense in which the loop *is* a difference equation manufactured from a differential one.

As a loop:

```
T      = 80.0          # initial condition
T_a    = 20.0          # parameter
k      = 0.1           # parameter, per minute
h      = 1.0           # step size, minutes
t      = 0.0
while t < 30.0:
    rate = -k * (T - T_a)     # the right-hand side, at the present value
    T    = T + h * rate       # Euler's step: one rectangle of the integral
    t    = t + h
    record(t, T)
```

Ten lines, and every solver of a differential equation in this book is this loop with a better line eight. The loop is what "integrating the equation" means to a program: adding up rate times step, one step at a time, from the initial condition forward. There is nothing else to it.

A.4.3 Worked example 1: three steps by hand

Step size $h = 1$ minute, $k = 0.1$, $T_a = 20$, $T_0 = 80$.

- Step 1. Rate at 80 is $-0.1 \times 60 = -6$. New value $T_1 = 80 + 1 \times (-6) = 74$.
- Step 2. Rate at 74 is $-0.1 \times 54 = -5.4$. New value $T_2 = 74 - 5.4 = 68.6$.
- Step 3. Rate at 68.6 is $-0.1 \times 48.6 = -4.86$. New value $T_3 = 68.6 - 4.86 = 63.74$.

The closed form gives 74.29, 69.12 and 64.45 at those three minutes, so the loop is under the truth by 0.29, 0.52 and 0.71 degrees. The error grows with each step and stays small; both facts are typical. The reason the loop undershoots is visible in the rectangle: it holds the rate at its *starting* value for the whole sliver, the true rate eases during the sliver, so every rectangle is a little too tall and the loop cools the coffee a little too fast every minute.

A.4.4 Worked example 2: the same three minutes with a larger step

Now take $h = 5$ minutes and do one step. Rate at 80 is -6; new value $T_1 = 80 + 5 \times (-6) = 50$. The closed form gives 56.39 at minute five, so the error is 6.4 degrees – twenty times the error of the one-minute step at minute one, for a step five times longer, because a wide rectangle misses much more of the curve's bend than a narrow one. Take $h = 10$ and one step lands on $80 - 60 = 20$, the room temperature, where the exact curve reads 42.07. Take $h = 25$ and one step gives $80 + 25 \times (-6) = -70$: the coffee has overshot the room by ninety degrees and would, on the next step, shoot back above it. The loop has not merely become inaccurate; it has become unstable, and it does so for this equation whenever $h > 2/k$, here twenty minutes.

Figure A.1 draws the three runs beside the exact curve.

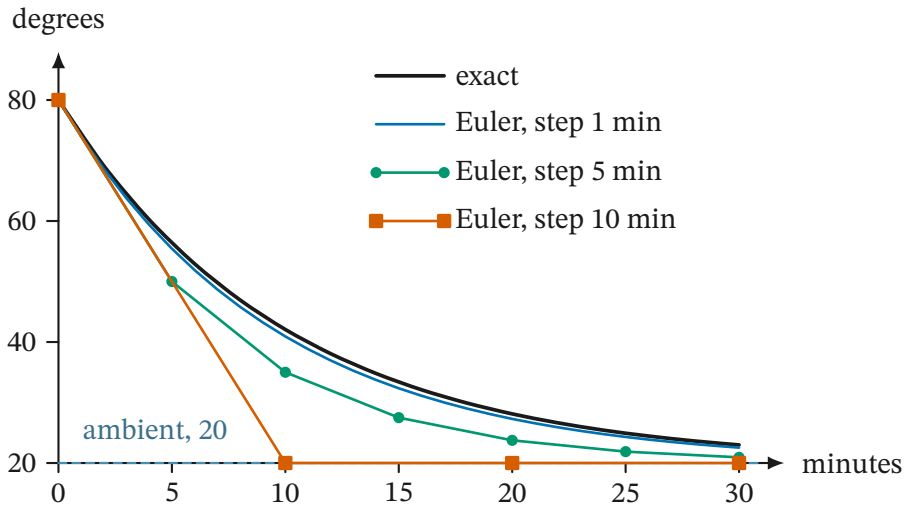


Figure A.1: The cooling curve from the same equation, initial condition and parameters, computed with Euler's step at three step sizes, beside the closed-form trajectory.

At a one-minute step the loop's curve lies under the exact one by less than a degree everywhere. At five minutes it visibly undershoots and then approaches the room from below. At ten minutes it falls to the room in one step and stays there: right about where the coffee ends up,

wrong about everything in between. **What the step size buys and costs**, in one sentence: a smaller h gives a trajectory closer to the truth, and a run of the same length takes proportionally more steps. Halve the step and you double the work; for Euler's method you also roughly halve the error at the end of the run.

A.4.5 Worked example 3: sampled time, with the sensor's period as the step

A twin stepping the coffee equation does not get to choose its clock freely: the thermometer reports every T_s seconds, and the twin's prediction is compared against each reading as it arrives. The natural choice is to make the sensor's period the step, $h = T_s$, so that one loop iteration produces one predicted value for each reported one. With a thermometer reporting every 30 seconds, $h = 0.5$ minutes, and the first two steps are $T_1 = 80 + 0.5 \times (-6) = 77$, then the rate at 77 is -5.7 and $T_2 = 77 - 2.85 = 74.15$, against a closed-form 74.29 at minute one: an error of 0.14, half the one-minute step's, as the step-size rule predicts. A loop run this way is in **sampled time**: one step per reading, so that the predicted value and the reported value arrive on the same clock and can be compared, which is the comparison a state estimator makes at every step.

A.4.6 Better Solvers

Euler's step is the simplest member of a large family, and the two members below are each one idea away from it. Both are shown at the same level of detail as Euler's step, on the same coffee, so the improvement is a number rather than a claim.

Heun's step: average the rate at both ends. The rectangle's fault is that it uses the rate at the start of the sliver only. A better estimate of the area is a trapezium: the average of the rate at the start and the rate at the end. The end rate is not known before the step is taken, so Heun's method takes an Euler step as a *guess* at the end value, reads the rate there, and averages the two:

$$T_{\text{guess}} = T_k + h\, f(T_k), \quad T_{k+1} = T_k + \frac{h}{2}\big(f(T_k) + f(T_{\text{guess}})\big) \tag{A.6}$$

Two rate evaluations per step instead of one, and in Equation A.6 the error at the end of a run shrinks with the *square* of the step rather than in proportion to it, so halving the step quarters the error. For the coffee with $h = 1$: the start rate is -6, the guess is 74, the rate there is -5.4, and $T_1 = 80 + 0.5 \times (-6 - 5.4) = 74.3$, against a closed form of 74.29 – an error of 0.01 where Euler's was 0.29. With $h = 5$: start rate -6, guess 50, rate there -3, $T_1 = 80 + 2.5 \times (-6 - 3) = 57.5$, against 56.39, where Euler gave 50. This is the two-stage member of the **Runge-Kutta** family; the widely used four-stage member samples the rate four times per step and its error shrinks with the fourth power of the step, so a step ten times longer gives the same accuracy at a quarter of the rate evaluations. The whole family exists because the fixed-step rectangle is expensive for the accuracy it buys [166].

```
rate1 = f(T)                    # rate at the start
guess = T + h * rate1           # an Euler step, used only to look ahead
rate2 = f(guess)                # rate at the guessed end
T     = T + h * (rate1 + rate2) / 2
```

The implicit step: use the rate at the end. Euler's step uses the rate at the start; the **implicit** (or backward) Euler step uses the rate at the *end* of the sliver, at the value it is about to compute:

$$T_{k+1} = T_k + h\, f(T_{k+1}) \tag{A.7}$$

The unknown T_{k+1} appears on both sides of Equation A.7, so each step is a small equation to solve rather than a formula to apply. For the cooling law it solves in one line: $T_{k+1} = T_k - h\,k\,(T_{k+1} - T_a)$ rearranges to $T_{k+1} = (T_k + h\,k\,T_a)/(1 + h\,k)$. For $h = 5$: $T_1 = (80 + 0.5 \times 20)/1.5 = 60$, against a closed form of 56.39 – less accurate than Heun, more accurate than Euler's 50, and, the reason it exists, *stable at any step size*. For $h = 25$, where Euler's loop overshot to -70, the implicit step gives $(80 + 2.5 \times 20)/3.5 = 37.1$: wrong by fifteen degrees, but on the right side of the room and heading the right way. For a non-linear equation the small equation is solved numerically at every step, which is the price; the reward is that the step can be as long as the accuracy allows rather than as short as stability demands.

Adaptive steps. A production solver does not use one h for the whole run. It takes a step, estimates its own error – Heun's method gives one for free, as the difference between the guess and the corrected value – and shortens the step where the curve bends or something switches, lengthening it again where the curve is gentle. Solvers of this kind take long steps while the dynamics are slow and short ones where they are fast or a switching instant must be located [169]. The user gives a tolerance instead of a step, and the solver spends its effort where the equation needs it.

Every simulator a twin might use offers these methods. This appendix uses Euler's step because it is the one you can do by hand, and every other solver is Euler with a better line eight. What none of them can remove is the cost of an equation that is *stiff*, which the Many Quantities section explains once the second quantity that causes it is on the table.

A.5 Many Quantities at Once

A twin rarely tracks one number. The coffee and the mug it sits in have two temperatures; a wind turbine's tower has a position and a velocity at every point along it; a battery has a charge, a temperature and an internal resistance. This section builds, one piece at a time, the notation that lets many quantities be handled as one: the vector, the matrix, the one operation between them, and then the system of differential equations they make possible.

A.5.1 Vectors: a record whose fields are numbers

Mathematics bundles the quantities of one system into a **vector**, and a vector is nothing more than a record whose fields are all numbers and are listed in a fixed order:

$$x = \begin{bmatrix} T_1 \\ T_2 \end{bmatrix}$$

The semicolon separates the fields, which mathematics stacks vertically and calls **components**; x is the whole record. The number of fields is the vector's **dimension**, n, and this book reserves x for the record of everything a model must remember – its **state** – so x has n components whenever the state has n quantities. The book sets a vector in the same plain letter as a single quantity; whether x has one component or a thousand is said in words where it matters. Two vectors of the same dimension are added field by field, and a vector is multiplied by a number by multiplying every field: $2\left[\begin{smallmatrix} 3 \\ 4 \end{smallmatrix}\right]$ is $\left[\begin{smallmatrix} 6 \\ 8 \end{smallmatrix}\right]$. Those two operations, and nothing else, are what "vector arithmetic" means in this book.

```
x = [T_1, T_2]              # a record: two named fields, in a fixed order
y = [a + b for (a, b) in zip(x, w)]     # addition, field by field
z = [2 * a for a in x]                  # a number times a vector
```

A.5.2 Matrices: a table of coefficients, and two things done to it

A **matrix** is a table of coefficients, m rows by n columns, and A_{ij} names the entry in row i and column j. Written on one line, rows are separated by semicolons and entries within a row by commas:

$$A = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}$$

is the two-by-two table whose first row is $2, 1$ and whose second is $0, 3$. A matrix with as many rows as columns is **square**, and the square matrix with ones down the diagonal and zeros elsewhere is the **identity**, I, which leaves every vector unchanged. Two things are done to a matrix in this book, and both are mechanical.

The transpose A^T is the table with rows and columns swapped: A_{ij} moves to row j, column i. For $A = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}$, $A^\mathsf{T} = \left[\begin{smallmatrix} 2 & 0 \\ 1 & 3 \end{smallmatrix}\right]$. A matrix equal to its own transpose is **symmetric**; every covariance matrix in the Probability section is.

The inverse A^{-1} is the table that undoes A: apply A and then A^{-1} and every vector comes back unchanged, $A^{-1} A\, x = x$. For a two-by-two there is a formula and the book needs no larger case:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{a\,d - b\,c} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \tag{A.8}$$

Swap the diagonal, negate the off-diagonal, divide by $a\,d - b\,c$. That divisor is the **determinant**, and when it is zero the inverse in Equation A.8 does not exist: the two rows of the table say the same thing in different words, so no table can tell their outputs apart. An inverse is what solving a linear system of equations means – if $A\,x = b$ then $x = A^{-1}\,b$ – and a zero determinant is the algebraic form of "these equations do not fix a unique answer".

A.5.3 Worked example 1: a transpose and an inverse, checked

$A = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}$. The transpose is $A^\mathsf{T} = \left[\begin{smallmatrix} 2 & 0 \\ 1 & 3 \end{smallmatrix}\right]$. The determinant is $2 \times 3 - 1 \times 0 = 6$. Swap the diagonal to get 3 and 2, negate the off-diagonal to get -1 and -0, divide by six: $A^{-1} = \left[\begin{smallmatrix} 0.5 & -0.1667 \\ 0 & 0.3333 \end{smallmatrix}\right]$. The check is in the next worked example, once multiplication is defined; the lesson here is that both operations are pure table manipulation, with no judgement in them. Now try $\left[\begin{smallmatrix} 1 & 2 \\ 2 & 4 \end{smallmatrix}\right]$: the determinant is $1 \times 4 - 2 \times 2 = 0$, the second row is twice the first, and no inverse exists.

A.5.4 Multiplication: each output is a weighted sum of the inputs

The one operation this book needs between a matrix and a vector is $y = A\,x$. Its meaning fits in a sentence: **each output component is a weighted sum of the input components, and row i of the table holds the weights for output i.** As code, that is two nested loops:

```
for i in 0 .. m-1:            # one output per row
    y[i] = 0
    for j in 0 .. n-1:        # weighted sum across the row
        y[i] = y[i] + A[i][j] * x[j]
```

If you have ever computed a weighted average, or a dot product, or a dense layer of a neural network, you have run this loop. Two matrices are multiplied the same way, one column of the second at a time, and that is the only other product this book uses: $A\,B$ is the table whose column

j is A applied to column j of B. Order matters – $A\,B$ and $B\,A$ are different tables in general – which is why every equation in this appendix is careful about which side a matrix sits on.

A.5.5 Worked example 2: a two-by-two times a vector, and the inverse checked

$A = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}$ and $x = \begin{bmatrix} 4 \\ 5 \end{bmatrix}$. Output component one is row one against x: $2\times4+1\times5 = 13$. Output component two is row two against x: $0\times4+3\times5 = 15$. So $y = \begin{bmatrix} 13 \\ 15 \end{bmatrix}$. Trace the loop: $i = 0$ runs j over 0 and 1 and accumulates 8 then 13; $i = 1$ accumulates 0 then 15. Four multiplications and two additions, which is n^2 multiplications for a square table – a number to remember when the section turns to cost.

Now the check the previous example promised. Apply $A^{-1} = \begin{bmatrix} 0.5 & -0.1667 \\ 0 & 0.3333 \end{bmatrix}$ to $y = \begin{bmatrix} 13 \\ 15 \end{bmatrix}$: row one gives $0.5 \times 13 - 0.1667 \times 15 = 6.5 - 2.5 = 4$, row two gives $0 \times 13 + 0.3333 \times 15 = 5$. The original $x = \begin{bmatrix} 4 \\ 5 \end{bmatrix}$ comes back, which is what "undoes" means.

A.5.6 Systems of Differential Equations

Two equations, each mentioning the other's variable. Put the coffee in a heavy ceramic mug. Now the coffee, at T_1, loses heat to the mug, at T_2, and the mug loses heat to the room. There are two quantities and two equations, and neither equation can be solved on its own, because each one's rate depends on the other one's value:

$$\frac{dT_1}{dt} = -k_1\,(T_1 - T_2), \quad \frac{dT_2}{dt} = k_2\,(T_1 - T_2) - k_3\,(T_2 - T_a)$$

The first says the coffee cools in proportion to how much hotter it is than the mug. The second says the mug warms in proportion to the same difference, with its own constant k_2 because a mug's heat capacity is not a coffee's, and cools to the room through k_3. Two equations that share variables are a **system of differential equations**, and a system is the ordinary case: almost nothing in a real asset changes on its own.

Excess over the room removes the constant. Measure both temperatures as excess over the room, $y_1 = T_1 - T_a$ and $y_2 = T_2 - T_a$. Since T_a is constant the rates are unchanged, and the room drops out of both equations: $dy_1/dt = -k_1\,(y_1 - y_2)$ and $dy_2/dt = k_2\,(y_1 - y_2) - k_3 * y_2$. Now every rate is a weighted sum of the two present values and nothing else – a **linear system**.

One matrix holds all the weights. Collect the two excesses into a vector $y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$ and the weights into a table, row one for the coffee's rate and row two for the mug's, and the pair of equations is one line:

$$\frac{dy}{dt} = A\,y, \quad A = \begin{bmatrix} -k_1 & k_1 \\ k_2 & -(k_2 + k_3) \end{bmatrix} \tag{A.9}$$

Equation A.9 is the one-quantity cooling law with a matrix where $-k$ used to be, and the derivative of a vector is the vector of derivatives, component by component. Row one of A multiplied into y gives $-k_1\,y_1 + k_1\,y_2$, which is the coffee's equation; row two gives the mug's. Nothing has been added and nothing lost; the matrix is a filing system for the coefficients.

Euler's step for a vector. The loop of the previous section applies unchanged, with the multiplication above in place of the scalar product: $y_{k+1} = y_k + h\,A\,y_k$. One matrix-times-vector per step – the two nested loops – inside the one loop that steps time.

```
y = [60.0, 0.0]                        # coffee excess, mug excess
A = [[-0.1, 0.1], [0.3, -0.5]]
h = 1.0
while t < 30.0:
```

```
rate = matvec(A, y)                # the two nested loops
y    = [y[i] + h * rate[i] for i in 0 .. 1]
t    = t + h
```

A second-order equation is a two-equation system. A vibrating mass obeys $d^2x/dt^2 = -(k_s/m_s)\,x$: its acceleration pulls it back in proportion to its displacement. Introduce the velocity $v = dx/dt$ as a second state, and the one second-order equation becomes two first-order ones, $dx/dt = v$ and $dv/dt = -(k_s/m_s)\,x$, which is the system $\frac{d}{dt}\left[\begin{smallmatrix} x \\ v \end{smallmatrix}\right] = \left[\begin{smallmatrix} 0 & 1 \\ -(k_s/m_s) & 0 \end{smallmatrix}\right]\left[\begin{smallmatrix} x \\ v \end{smallmatrix}\right]$. Every higher-order equation is handled this way, which is why a solver needs only first-order systems and why "state" so often means position *and* velocity.

A.5.7 Worked example 3: two Euler steps of the coupled system

Take $k_1 = 0.1$, $k_2 = 0.3$, $k_3 = 0.2$ per minute, so $A = \left[\begin{smallmatrix} -0.1 & 0.1 \\ 0.3 & -0.5 \end{smallmatrix}\right]$, and start with the coffee at 80 and the mug at room temperature: $y_0 = \left[\begin{smallmatrix} 60 \\ 0 \end{smallmatrix}\right]$.

- Step 1. The rate is $A\,y_0$: row one gives $-0.1 \times 60 + 0.1 \times 0 = -6$, row two gives $0.3 \times 60 - 0.5 \times 0 = 18$. The coffee is cooling at 6 degrees a minute and the mug warming at 18, fast, because a cold mug against hot coffee is a large excess through a large k_2. With $h = 1$, $y_1 = \left[\begin{smallmatrix} 60-6 \\ 0+18 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 54 \\ 18 \end{smallmatrix}\right]$: coffee 74, mug 38.
- Step 2. The rate at y_1 is row one $-0.1\times54+0.1\times18 = -3.6$, row two $0.3\times54-0.5\times18 = 7.2$. So $y_2 = \left[\begin{smallmatrix} 54-3.6 \\ 18+7.2 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 50.4 \\ 25.2 \end{smallmatrix}\right]$: coffee 70.4, mug 45.2.

The coffee's rate fell from -6 to -3.6 in one minute, far faster than the lone cup's did, because the mug it is losing heat to is catching up with it; the mug's rate fell from 18 to 7.2 for the same reason. The reasoning behind each move is the same as in the one-quantity case, done once per row: evaluate the equation at the present record, multiply by the step, add.

A.5.8 State-Space Equations

The form. A system of differential equations becomes a **state-space model** when three more things are named: an **input** u, the quantities done to the system from outside – a heater's power, a valve setting, a command – an **output** z, the quantities a sensor actually reports, and the tables that say how each enters. For a linear system the whole model is two lines on one:

$$\frac{dx}{dt} = A\,x + B\,u, \quad z = C\,x \tag{A.10}$$

In Equation A.10, A is the table of the system's own dynamics, as in the coupled cooling; B says which states an input pushes on and how hard; C says which combination of states a sensor sees. For the coffee and mug with a small heater under the mug delivering u watts, and a thermometer in the coffee only: $x = \left[\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix}\right]$, $B = \left[\begin{smallmatrix} 0 \\ b \end{smallmatrix}\right]$ where b is the degrees per minute one watt buys the mug, and $C = \left[\begin{smallmatrix} 1 & 0 \end{smallmatrix}\right]$, because the thermometer reads the first state and nothing of the second. A non-linear system uses the same shape with functions in place of the tables, $dx/dt = f(x, u, \theta)$ and $z = g(x)$, and the Markov section adds the noise terms that turn this into the model a state estimator uses.

The discrete form. A twin runs the state-space model as a difference equation, in sampled time. Euler's step with h equal to the sensor period gives it directly:

$$x_{k+1} = F\,x_k + G\,u_k, \quad z_k = C\,x_k, \quad F = I + hA, \quad G = hB \tag{A.11}$$

Equation A.11 is the **discrete-time state-space model**, and every symbol in it is a table computed once before the loop starts. The step from the continuous form to this one is called

discretisation, and it is what turns a physical model into a process that produces a sample per period [190]. Euler's $F = I + hA$ is the simplest discretisation; a solver's better step gives a better F, and for a linear system there is an exact one.

What the form is for. Three things, and all three run through this book. First, *uniformity*: a solver, an estimator or a controller written against (A, B, C) works for any linear system, whatever the physics, which is what lets one estimator serve a coffee cup and a turbine tower. Second, *the estimator itself*: a state estimator needs to know how the state moves on its own, how the input moves it, and what the sensor would report for a given state, and those are exactly A, B and C – a physics-based twin of a floating wind turbine, for instance, runs a Kalman filter over a linear model of the structure to estimate the structural states and the section loads along the tower [164]. Third, *the unmeasured*: C is usually a short, wide table, because a sensor sees a few combinations of many states, and the whole point of carrying x rather than z is to reason about the states the sensor does not see.

```
F = I + h * A                 # computed once, before the loop
G = h * B
x = x_0
for each sensor period k:
    x = matvec(F, x) + matvec(G, u[k])     # predicted state
    z = matvec(C, x)                       # what the sensor should report
    record(k, x, z)
```

A.5.9 Worked example 4: the coffee and mug in state-space form

With $h = 1$ minute and $A = \left[\begin{smallmatrix} -0.1 & 0.1 \\ 0.3 & -0.5 \end{smallmatrix}\right]$, $F = I + hA = \left[\begin{smallmatrix} 0.9 & 0.1 \\ 0.3 & 0.5 \end{smallmatrix}\right]$. Apply F to $x_0 = \left[\begin{smallmatrix} 60 \\ 0 \end{smallmatrix}\right]$: row one $0.9 \times 60 + 0.1 \times 0 = 54$, row two $0.3 \times 60 + 0.5 \times 0 = 18$, so $x_1 = \left[\begin{smallmatrix} 54 \\ 18 \end{smallmatrix}\right]$ – the same answer as Euler's step in the previous worked example, because F *is* Euler's step written as a table. Now switch the heater on at $u = 20$ watts with $b = 0.05$ degrees per minute per watt: $G = hB = \left[\begin{smallmatrix} 0 \\ 0.05 \end{smallmatrix}\right]$, and $Gu = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$, so $x_1 = \left[\begin{smallmatrix} 54 \\ 19 \end{smallmatrix}\right]$: the heater adds one degree a minute to the mug and nothing directly to the coffee. The output is $z_1 = C\,x_1 = 54$ – the thermometer reports the coffee's excess and knows nothing of the mug's, which a twin would then have to infer.

A.5.10 Stiff Equations for Digital Twins

The word **stiff** describes a system, not a solver, and it appears the moment a system has two quantities that move at very different speeds. Put a thermocouple in the coffee: the probe tip settles to the coffee's temperature with its own k of perhaps 100 per minute, while the coffee itself cools with $k = 0.1$. Euler's step is stable for the probe only below 2/100, or about a second, so a fixed-step Euler loop must take a step every second for a whole hour to follow a probe that settled in the first three – a thousand times the work the coffee curve needs on its own account. A system is stiff when it holds a very fast process and a very slow one at once and the fast one is over almost immediately, so that stability, not accuracy, dictates the step.

The coupled coffee and mug are mildly stiff already: the Modes subsection below shows that the system has a fast part with a time constant of under two minutes and a slow part with one of nearly thirty, so Euler's step must stay under about three and a half minutes for the fast part while the slow part would be happy with ten. In a twin, stiffness is the ordinary case rather than the exception – an electrical transient inside a thermal process, a controller switching inside a slow plant, a probe inside a vessel – and it is the reason the implicit and adaptive solvers of the previous section exist. If a model of yours runs inexplicably slowly at a fixed step, stiffness is the first thing to suspect, and the remedy is an implicit step, which the previous section showed is stable at any step size, or a solver that adapts.

A.5.11 Dimension, and cost growing with it

The two loops cost $m\,n$ multiplications, n^2 for a square table. The coffee and mug have $n = 2$ and four multiplications a step. A model of a turbine tower that keeps a position and a velocity at five hundred points along it has $n = 1000$ and a million multiplications a step. A finite-element model of the same tower keeps three displacements at each of a few hundred thousand mesh points and has n around a million, and a million squared is 10^{12} multiplications a step. It is the same loop with a million-row table. The table is mostly zeros, because each mesh point touches only its neighbours, and every real simulator exploits that; the point stands that cost grows with dimension, and faster than dimension. The What a Solver and a Simulator Do section turns this into the one number that decides whether a model can live inside a twin.

A.5.12 Modes: the shapes a system prefers to move in

The idea. Pluck a guitar string and it vibrates in a shape: the whole string bowing out and back, at one pitch. Touch it lightly at the midpoint and pluck again: two half-length bows moving oppositely, at twice the pitch. Each shape, with its own frequency, is a **mode** of the string – a pattern of motion the structure prefers, in which every point moves in step at one **natural frequency**. Modal analysis of a monitored structure extracts exactly these from vibration recordings: the natural frequencies, which carry most of the structure's vibration energy, and the mode shapes, the spatial patterns it vibrates in at each of them [191].

The mathematics, for a linear system. A mode of $dy/dt = A\,y$ is a vector v that the matrix does not turn, only stretches: applying A to v gives back v times a number λ. If the system starts in such a shape it stays in it, every component scaled by the same $\exp(\lambda\,t)$:

$$A\,v = \lambda\,v, \quad y(t) = e^{\lambda t}\,v \tag{A.12}$$

The pair in Equation A.12 is the mode shape v and its rate λ; a negative λ is a decay with time constant $-1/\lambda$, and for a vibrating system λ comes as a pair whose imaginary part is the natural frequency. Any starting state is a mix of the modes, and the trajectory is that mix with each mode decaying or oscillating at its own rate, which is why a handful of modes describes most of what a large linear system does. Finding v and λ from A is a standard computation every numerical library provides; for a two-by-two it is a quadratic equation, and the worked example does it.

Natural frequency, for a vibrating mass. The single mass on a spring of the previous subsection obeys $d^2x/dt^2 = -(k_\mathrm{s}/m_\mathrm{s})\,x$, and the motion $x(t) = a\,\cos(\omega\,t)$ satisfies it exactly when $\omega^2 = k_\mathrm{s}/m_\mathrm{s}$:

$$\omega = \sqrt{k_\mathrm{s}/m_\mathrm{s}}, \quad f_\mathrm{n} = \frac{\omega}{2\pi} \tag{A.13}$$

Equation A.13 gives the **natural frequency** in radians per second and in cycles per second: stiffer springs raise it, heavier masses lower it. A tall structure is many masses and many springs, and each of its modes has such a frequency; the lowest is the whole structure swaying, and a wind-turbine tower's is typically a fraction of a cycle per second.

A.5.13 Worked example 5: the two modes of the coffee and mug

For $A = \left[\begin{smallmatrix} -0.1 & 0.1 \\ 0.3 & -0.5 \end{smallmatrix}\right]$, the condition $A\,v = \lambda\,v$ has a non-zero solution only when the determinant of $A - \lambda\,I$ is zero: $(-0.1-\lambda)\,(-0.5-\lambda) - 0.1\times 0.3 = 0$, which multiplies out to $\lambda^2 + 0.6\,\lambda + 0.02 = 0$. The quadratic formula gives $\lambda = \left(-0.6 \pm \sqrt{0.36 - 0.08}\right)/2$, so $\lambda_1 = -0.0354$ and $\lambda_2 = -0.5646$ per minute. Time constants $-1/\lambda$: 28.2 minutes and 1.77 minutes. For each, the shape comes from row one of $A\,v = \lambda\,v$: $-0.1\,v_1 + 0.1\,v_2 = \lambda\,v_1$, so $v_2 = (\lambda + 0.1)\,v_1/0.1$. Taking $v_1 = 1$: the

slow mode is $v = \begin{bmatrix} 1 \\ 0.646 \end{bmatrix}$ and the fast mode $v = \begin{bmatrix} 1 \\ -4.646 \end{bmatrix}$. Check the slow one against row two: $0.3 \times 1 - 0.5 \times 0.646 = -0.023$, and $\lambda_1\, 0.646 = -0.0229$; they agree.

Read the two shapes. The slow mode has both excesses positive with the mug at 65 per cent of the coffee's: the two cooling *together* to the room over half an hour. The fast mode has them opposite in sign: the coffee and mug *equalising* with each other in under two minutes. Every trajectory of this system is a mix of the two, and after a few minutes only the slow one is left – which is what the eye sees in a real mug of coffee, and what the Stiff Equations subsection meant by a fast part and a slow part.

A.5.14 Worked example 6: a natural frequency, and two masses

A 2 kilogram mass on a spring of 50 newtons per metre has $\omega = \sqrt{50/2} = 5$ radians per second, so $f_\mathrm{n} = 5/(2\,\pi) = 0.80$ cycles per second and a period of 1.26 seconds. Now two equal masses between three equal springs, wall to mass to mass to wall. Each mass feels its own spring and the shared one: $m_\mathrm{s}\, d^2x_1/dt^2 = -2\,k_\mathrm{s}\, x_1 + k_\mathrm{s}\, x_2$ and the mirror image for x_2. Try the two obvious shapes. In the **in-phase** shape $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ the masses move together, the shared spring never stretches, each mass feels one spring, and $\omega_1 = \sqrt{k_\mathrm{s}/m_\mathrm{s}} = 5$: substituting $x_2 = x_1$ into the first equation gives $-k_\mathrm{s}\, x_1$ on the right, as claimed. In the **anti-phase** shape $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$ the masses move oppositely, the shared spring stretches twice as much, and substituting $x_2 = -x_1$ gives $-3\,k_\mathrm{s}\, x_1$, so $\omega_2 = \sqrt{3\,k_\mathrm{s}/m_\mathrm{s}} = 8.66$ radians per second. Two masses, two modes, two natural frequencies; a tower with a million states has a million, and a reduced model keeps the first few.

A.6 Probability and Distributions

A.6.1 Probability: from counting to a number between zero and one

A pump has been logged every hour for a year. Of its 8,760 hours, it was off for 5,694, on for 2,628 and faulted for 438. Ask "what is the chance the pump is on at a randomly chosen hour" and the classical answer is a ratio: the number of hours that satisfy the condition over the number of hours there are, $2628/8760 = 0.3$. That ratio is a **probability**: a number between zero and one, where zero means never and one means always. An **event** is a condition on the outcome – "the pump is on", "the reading is above 70" – and the probability of an event is the fraction of outcomes in which it holds, when every outcome is equally likely, or the fraction of the time it has held, when the outcomes are observed rather than counted. Three rules follow from the ratio and are all the probability calculus this book uses:

- The probabilities of all the outcomes sum to one: $0.65 + 0.3 + 0.05 = 1$, because the pump is always in *some* state.
- Two events that cannot both happen – **mutually exclusive** – have probabilities that add: $P(\text{onorfaulted}) = 0.3 + 0.05 = 0.35$.
- The probability that an event does not happen is one minus the probability that it does: $P(\text{notfaulted}) = 1 - 0.05 = 0.95$.

A.6.2 Worked example 1: three probabilities from one log

From the log above, the probability that the pump is not off is $1 - 0.65 = 0.35$, which is the same as $P(\text{onorfaulted})$, as it must be: "not off" and "on or faulted" are the same event said two ways. The probability that it is off *or* on is $0.65 + 0.3 = 0.95$, which is the complement of faulted. And the probability that it is on *and* faulted is zero, because one pump is in one state at a time; that is what mutually exclusive means, and it is why the second rule could add.

A.6.3 Conditional probability: "given"

Often the question is not "how likely is this" but "how likely is this, *given* that". Of a fleet of 1,000 bearings at the same age, 50 are running hot and 950 are normal. A vibration alarm has fired on 45 of the hot ones and, falsely, on 95 of the normal ones. Given that a bearing's alarm has fired, how likely is it hot? Restrict the count to the bearings that satisfy the condition – the 140 with an alarm – and count the hot ones among them: $45/140 = 0.32$. That is a **conditional probability**, written $P(\text{hot} \mid \text{alarm})$ and read "the probability of hot given alarm", and the general form is the ratio of the joint count to the conditioning count:

$$P(A \mid B) = \frac{P(A \text{ and } B)}{P(B)} \tag{A.14}$$

In Equation A.14, $P(A \text{and} B)$ is the probability that both events hold – $45/1000 = 0.045$ for hot and alarm – and $P(B)$ is the probability of the condition, $140/1000 = 0.14$. The division rescales the joint probability so that it is measured against the condition rather than against everything. The bar is read "given", and everything to its right is what you already know.

A.6.4 Bayes' rule: turning a conditional around

The same joint count can be conditioned the other way. $P(\text{alarm} \mid \text{hot})$ is the fraction of *hot* bearings whose alarm fired, $45/50 = 0.9$; that is a property of the sensor, and it is usually what a datasheet or a test bench gives you. What a twin needs is the reverse, $P(\text{hot} \mid \text{alarm})$ – what to believe about the bearing given what the sensor said. Writing the joint probability both ways, $P(A \text{and} B) = P(A \mid B)\,P(B)$ and $P(A \text{and} B) = P(B \mid A)\,P(A)$, and equating them gives the one relation every estimator in this book is built on:

$$P(A \mid B) = \frac{P(B \mid A)\,P(A)}{P(B)} \tag{A.15}$$

Equation A.15 is **Bayes' rule**, and the words attached to its symbols matter more than the symbols. With A the state and B the reading: $P(A)$ is what you believed about the state *before* the reading, the **prior**; $P(B \mid A)$ is how likely that reading would be *if* the state were A, the **likelihood**, a property of the sensor; $P(A \mid B)$ is what you believe *after* the reading, the **posterior**; and $P(B)$ is the total probability of the reading, found by adding $P(B \mid A)\,P(A)$ over every possible state, which is what makes the posteriors sum to one. The rule says: posterior is proportional to likelihood times prior. Every correction step in a state estimator is this line with the symbols dressed up.

```
prior      = {hot: 0.05, normal: 0.95}
likelihood = {hot: 0.9,  normal: 0.1}          # P(alarm | state)
unnorm     = {s: likelihood[s] * prior[s] for s in prior}
total      = sum(unnorm.values())                # P(alarm)
posterior  = {s: unnorm[s] / total for s in prior}
```

A.6.5 Worked example 2: Bayes' rule on the bearing, twice

Prior $P(\text{hot}) = 0.05$, $P(\text{normal}) = 0.95$, from the fleet counts. Likelihoods $P(\text{alarm} \mid \text{hot}) = 0.9$ and $P(\text{alarm} \mid \text{normal}) = 0.1$, from the sensor. The alarm fires. Top line for each state: hot, $0.9 \times 0.05 = 0.045$; normal, $0.1 \times 0.95 = 0.095$. Total $P(\text{alarm}) = 0.14$. Posterior $P(\text{hot} \mid \text{alarm}) = 0.045/0.14 = 0.32$ – the same 0.32 the direct count gave, which is the check that Bayes' rule is the counting argument in symbols. One alarm from a 90-per-cent sensor leaves the bearing more likely fine than hot, because hot bearings were rare to begin with; that is the prior doing its work, and it is why a twin diagnoses from a belief and not from a single reading.

A second alarm an hour later, taken as independent of the first given the bearing's state, starts from the new prior of 0.32: hot, $0.9 \times 0.32 = 0.29$; normal, $0.1 \times 0.68 = 0.068$; total 0.357; posterior $0.29/0.357 = 0.81$. Two alarms make it likely. The posterior of one step became the prior of the next, which is what the Markov section will call the belief.

A.6.6 Independence

Two events are **independent** when knowing one changes nothing about the other: $P(A \mid B) = P(A)$, or equivalently

$$P(A \text{ and } B) = P(A)\,P(B) \tag{A.16}$$

Equation A.16 is the test as well as the definition: compute both sides from the counts, and the events are independent only if they agree. The two alarms above were treated as independent *given* the bearing's state, which is what let the second update start from the first's posterior; a state estimator assumes the noise in the model's step and the noise in the sensor are independent of each other and of everything earlier, and that assumption is the one most often violated in practice, by a sensor whose error drifts rather than resets.

A.6.7 Worked example 3: two sensors, dependent or not

Two pressure sensors on the same pipe each fail on any given day with probability 0.01. If they fail independently, the probability that both fail today is $0.01 \times 0.01 = 0.0001$, one day in ten thousand, which is why a second sensor is bought. Now suppose both are fed from one power supply that fails on one day in two hundred, 0.005, taking both sensors with it. The probability that both fail is now *at least* 0.005, fifty times the independent figure, and the test fails: $P(\text{both}) \geq 0.005$ but $P(A)\,P(B) = 0.0001$. The second sensor was bought for independence and the wiring took it away. Independence is a fact about the world to be checked, not a property to be assumed because there are two of something.

A.6.8 From events to quantities: the density

A thermometer in the coffee at minute three does not read 64.45. It reads 64.3, or 64.7, or 64.5, and if you read it a hundred times it reads a hundred slightly different numbers. The reading is a **random quantity**: one you cannot compute, only describe by how likely each value is. For a quantity that takes one of a few values the description is a list – the pump's 0.65, 0.3, 0.05 – called a **probability mass function**. For a quantity that can take any value on a line, no single value has a probability of its own, and the description is a **density** $p(x)$: a non-negative function of the possible values whose *area* over an interval is the probability that the quantity lands in that interval.

$$P(\alpha \leq x \leq \beta) = \int_{\alpha}^{\beta} p(x)\,dx \tag{A.17}$$

Equation A.17 asks for the area under a curve between two points, which is the integral of The Integration section. That is all a density is – a curve whose areas are probabilities – and the one condition on it is that the whole area is one, because the quantity has to land somewhere. The three rules of the first subsection carry over with sums replaced by integrals: areas over non-overlapping intervals add, and the area outside an interval is one minus the area inside.

Mean, variance, standard deviation. Two numbers summarise a distribution. The **mean** μ is where it balances, and the **variance** σ^2 is how far it spreads about that point, measured as

the average squared distance. For a list they are weighted sums; for a density the sums become integrals:

$$\mu = \int x\, p(x)\, dx, \quad \sigma^2 = \int (x-\mu)^2\, p(x)\, dx \tag{A.18}$$

Both integrals in Equation A.18 run over every value the quantity can take. The variance carries the quantity's units squared, which is awkward to think in, so its square root, the **standard deviation** σ, is what gets quoted: a thermometer "good to half a degree" has $\sigma = 0.5$ degrees and a variance of 0.25 degrees squared. Every uncertainty figure in this book is a standard deviation unless it says variance. For a vector of quantities the variance becomes a table, the **covariance matrix** P: the two variances on the diagonal, and off it a number saying how the two move together, positive if one runs high when the other does, zero if knowing one says nothing about the other. It is symmetric, and every P, Q and R a state estimator carries is one.

```
mean = sum(x * p(x) for x in values)                    # a list
var  = sum((x - mean)**2 * p(x) for x in values)
sd   = sqrt(var)
# for a density, replace each sum by the integral over the line
```

A.6.9 Distributions: the two shapes this book uses

A **distribution** is a named family of densities, fixed by a few numbers. Two do all the work here.

The uniform distribution. Every value in an interval from a to b is equally likely and nothing outside it is possible. The density is flat, $p(x) = 1/(b-a)$ on the interval, so that its area is one, and zero elsewhere. Its mean is the midpoint, $(a+b)/2$, and its variance is $(b-a)^2/12$, so its standard deviation is about 0.29 of the width. It is the distribution of a reading you know only to within a rounding step, and of a parameter you know only to within a range.

The normal, or Gaussian, distribution. The bell curve, written $\mathcal{N}(\mu, \sigma^2)$:

$$p(x) = \frac{\exp\!\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)}{\sqrt{2\pi\sigma^2}} \tag{A.19}$$

Equation A.19 is fixed completely by its mean and variance: give those two numbers and the whole curve is known. Three areas under it are worth memorising: about 68 per cent of the probability lies within one standard deviation of the mean, 95 per cent within two, and 99.7 per cent within three. It is the default description of sensor noise, and for a reason rather than a habit: a disturbance made up of many small independent causes tends towards a bell curve whatever the causes' own shapes, and it is the only distribution under which a belief can be carried as a mean and a covariance alone, which is what makes a state estimator tractable. One more fact about it is used constantly: **a linear rule preserves it**. Multiply a Gaussian quantity by a constant a and add a constant b, and the result is Gaussian with mean $a\,\mu + b$ and variance $a^2\,\sigma^2$; the mean goes through the rule, the standard deviation is scaled by the multiplier, and the offset does not touch it. Where the Gaussian fails – a sensor that sticks, a quantity that cannot go negative, a process with rare large jumps – a state estimator built on it fails with it, and the book says so where it arises.

A.6.10 Worked example 4: a rounded reading as a uniform density

A sensor reports a reading rounded to the nearest ten units, so a displayed 15 means the true value is somewhere between 10 and 20 with no value more likely than another. The density is

$p(x) = 0.1$ on the interval from 10 to 20, and its area is $0.1 \times 10 = 1$ as required. The probability that the true value lies between 12 and 15 is the area of the rectangle over that span, $0.1 \times 3 = 0.3$. The mean is the midpoint, $\mu = 15$, which the first integral confirms: $\int_{10}^{20} 0.1\, x\, dx = 0.1 \times (400 - 100)/2 = 15$. The variance is the second integral, $\int_{10}^{20} (x - 15)^2 \times 0.1\, dx$; substitute $u = x - 15$, integrate u^2 from -5 to 5 to get 250/3, and multiply by 0.1 to get $\sigma^2 = 8.33$, which is $(20-10)^2/12$ as the family's formula says. The standard deviation is $\sigma = 2.89$ units, about a third of the interval's width.

A.6.11 Worked example 5: a thermometer as a Gaussian

The thermometer's reading of the coffee at minute three is Gaussian with mean 64.45 and $\sigma = 0.5$ degrees. The probability that a single reading lies within half a degree of the truth – between 63.95 and 64.95 – is the one-standard-deviation area, about 0.68; within one degree, about 0.95; within one and a half, 0.997. So roughly one reading in twenty is more than a degree off, and one in three hundred more than a degree and a half. Now push the reading through one Euler step of the cooling equation, $T_1 = (1 - h\,k)\,T_0 + h\,k\,T_a$, a linear rule with multiplier $a = 0.9$ and offset $b = 2$ at $h = 1$. By the linear rule fact, the predicted T_1 is Gaussian with mean $0.9 \times 64.45 + 2 = 60.0$ and standard deviation $0.9 \times 0.5 = 0.45$ degrees. The step *shrank* the uncertainty, because the cooling law pulls every trajectory toward the room and so pulls nearby trajectories together. Convert the reading to Fahrenheit instead, $F = 1.8\,T + 32$: the mean becomes 148.0 and the standard deviation $1.8 \times 0.5 = 0.9$; the offset of 32 moved the mean and left the spread alone.

A.7 State Machines with Probabilities: The Markov Chain

A.7.1 Boxes and arrows

Not every quantity a twin tracks is a number that slides. A pump is off, on, or faulted, and it jumps between those. You already know the picture for that: a **finite state machine** – boxes for the states, arrows for the transitions, exactly one box active at a time, and on each arrow the event that fires it. A watering controller is one; so is the lifecycle of a network connection. A finite state machine is a differential equation's discrete cousin: instead of "given where you are, here is how fast you move", it says "given where you are, here is where you go next".

A.7.2 Put a probability on every arrow

A twin usually cannot say *which* arrow fires next, only how likely each one is over the next interval. Write that likelihood on the arrow and the picture is a **Markov chain**. Figure A.2 draws one for the pump, with a step of one hour.

Read Figure A.2 from each state in turn. An idle pump stays idle with probability 0.7 and starts with 0.3. A running pump stops with 0.4, keeps running with 0.58, and fails with 0.02. A faulted pump is repaired and returned to idle with 0.1 and stays faulted with 0.9. The numbers on the arrows leaving any one state sum to one, because the pump will be in *some* state an hour from now.

The same picture as a table is the **transition matrix** Π: one row per present state, one column per next state, and Π_{ij} the probability of going from state i to state j in one step. Ordering the states Off, On, Faulted:

$$\Pi = \begin{bmatrix} 0.7 & 0.3 & 0 \\ 0.4 & 0.58 & 0.02 \\ 0.1 & 0 & 0.9 \end{bmatrix} \tag{A.20}$$

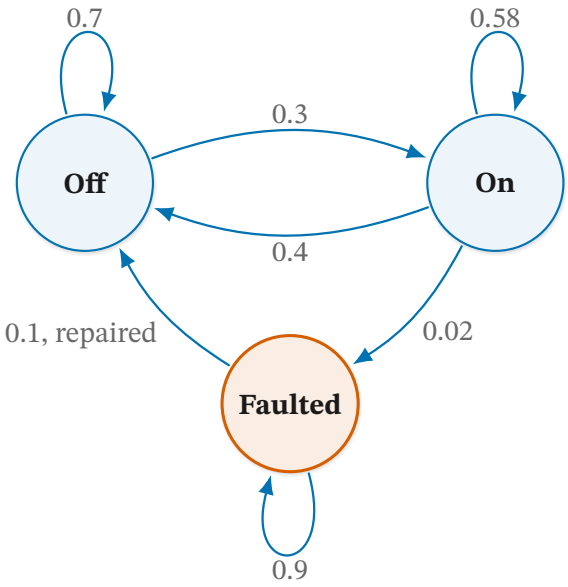


Figure A.2: A three-state pump as a Markov chain. Each arrow carries the probability of taking it in the next hour, including the arrow from a state back to itself.

Each row of Equation A.20 sums to one – a **row-stochastic** matrix – and that is the whole of what the chain knows. The table *is* the model; the picture is its documentation.

A.7.3 Worked example 1: from a state machine to its matrix

Start with a state machine you would write anyway: three states, and the transitions Off to On (a start command), On to Off (a stop), On to Faulted (a failure), Faulted to Off (a repair). Ask, for each state and each hour, how often each transition fires. Operating logs for this pump say it runs about thirty per cent of its idle hours, so $\Pi_{12} = 0.3$ and the rest of the row, $\Pi_{11} = 0.7$, is staying put. It stops in forty per cent of its running hours and fails in two per cent, so $\Pi_{21} = 0.4$, $\Pi_{23} = 0.02$, and staying running takes the rest, $\Pi_{22} = 0.58$. Repair takes about ten hours on average, so a faulted pump leaves that state with probability 0.1 per hour, to Off; $\Pi_{31} = 0.1$, $\Pi_{33} = 0.9$, and $\Pi_{32} = 0$ because a repaired pump is never returned straight into service. Three transitions that the state machine did not have – the self-loops – appeared because every row must sum to one; that is the one thing the probabilistic version adds to the picture you started from.

A.7.4 The belief, and prediction as one multiplication

If you knew the pump's state you would write it as a row with a one in that state's column and zeros elsewhere. If you do not know it, you write how likely each state is: a **belief**, b_k, a row of probabilities that sums to one. To predict the belief one step ahead, ask: how much probability flows into each next state? Into state j flows, from every present state i, the probability of being in i times the probability of going from i to j. That sentence is the weighted-sum loop of the Many Quantities section with the row on the left:

$$b_{k+1} = b_k \, \Pi \tag{A.21}$$

One matrix multiplication per step, and Equation A.21 is the discrete form of the Euler loop: present belief in, next belief out, the table in between. With a row on the left the two nested loops run column by column:

```
b = [1.0, 0.0, 0.0]                    # certain: Off
for each hour:
    b_next = [0.0, 0.0, 0.0]
    for j in 0 .. 2:                   # one entry per next state
        for i in 0 .. 2:               # probability flowing in from state i
            b_next[j] = b_next[j] + b[i] * Pi[i][j]
    b = b_next
```

A.7.5 Worked example 2: two prediction steps, and the leak

The pump was seen idle at hour zero: $b_0 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$. One step: $b_1 = b_0 \Pi$ picks out row one of the table, $\begin{bmatrix} 0.7 & 0.3 & 0 \end{bmatrix}$. The belief is no longer certain; the pump is probably still idle but may have started. Second step, column by column. Off: $0.7 \times 0.7 + 0.3 \times 0.4 + 0 \times 0.1 = 0.61$. On: $0.7 \times 0.3 + 0.3 \times 0.58 + 0 \times 0 = 0.384$. Faulted: $0.7 \times 0 + 0.3 \times 0.02 + 0 \times 0.9 = 0.006$. So $b_2 = \begin{bmatrix} 0.61 & 0.384 & 0.006 \end{bmatrix}$, which sums to one. The largest entry fell from 1 to 0.7 to 0.61 in two hours: **certainty leaks away with every step** you predict without looking. Run the multiplication on and the belief settles to a fixed row that no longer depends on where the pump started, at which point the twin knows only what the logs knew. This is a twin's staleness in arithmetic: a twin that has not heard from its pump for a while holds a belief, and the belief is worth less the longer the silence.

A.7.6 The Markov property, in words and in one line

Everything above assumed one thing: that the next state depends on the present state and the present input, and on nothing earlier.

$$P(x_{k+1} \mid x_k, u_k, x_{k-1}, u_{k-1}, \dots) = P(x_{k+1} \mid x_k, u_k) \tag{A.22}$$

Equation A.22 is the **Markov property**. The history to the left of the bar may be as long as you like; only the last state and input on it matter. It is the reason a belief is a single row and not a log of every past reading, and the reason the loop of the earlier sections can hold one value of T rather than its whole history: the state is *defined* as whatever the future depends on. If the future depends on something the state omits – how long the pump has already been running, say – the property fails, and the repair is to add that quantity to the state, not to abandon the chain.

A.7.7 The transition density and the noisy state-space model

For a quantity that slides rather than jumps, the rows of the table become densities. The **transition density** $p(x_{k+1} \mid x_k, u_k)$ says how likely each next value is given the present value and input; it is one row of Π, with a curve in place of a list. Where does that curve come from? From the state-space model of the Many Quantities section, with noise added:

$$x_{k+1} = f(x_k, u_k, \theta) + w_k, \quad z_k = g(x_k) + v_k \tag{A.23}$$

Equation A.23 is the discrete state-space model with two new terms. The first line is the step rule – $F x_k + G u_k$ for a linear system, or any other rule that gives the next state – with a **process noise** w_k added for everything the rule leaves out. The second is the output rule with

a **measurement noise** v_k for everything the sensor gets wrong. For the coffee, $f(T_k) = T_k - h\,k\,(T_k - T_a)$, $g(T_k) = T_k$, and if w_k is Gaussian with standard deviation 0.3 degrees, the transition density is a bell curve centred on the Euler prediction with $\sigma = 0.3$. The whole apparatus of a state estimator – and the formal descriptions of a twin as a pair of coupled dynamical systems carried by a probabilistic graphical model [171] – is this one line, chosen so that the Markov property holds.

A.7.8 A chain seen through a noisy sensor

The twin does not see the pump's state. It sees a current sensor that reports "drawing current" or "no current", and the sensor is imperfect: it reports current with probability 0.95 when the pump is On, 0.05 when Off (leakage and noise), and 0.3 when Faulted (a stalled motor draws some). A Markov chain observed through a sensor like that is a **hidden Markov model**, and Figure A.3 draws it.

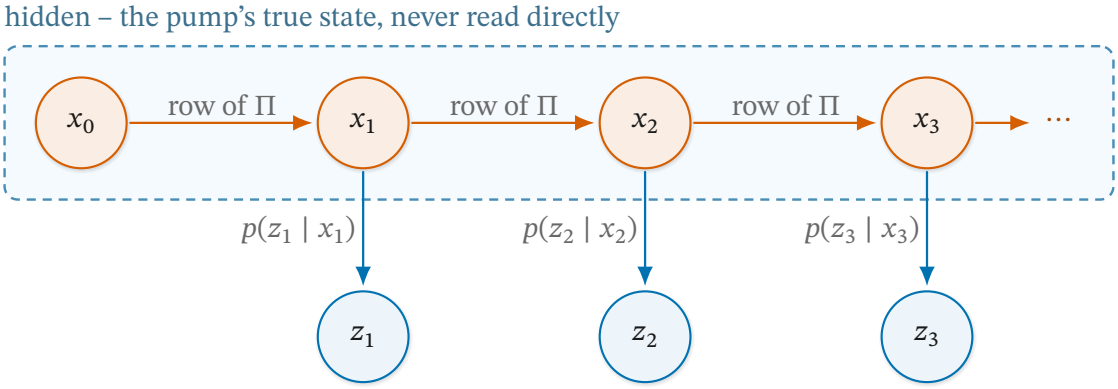


Figure A.3: The pump chain seen through its current sensor. The states along the top are hidden; each reading below depends on its own state alone.

In Figure A.3 every arrow along the top goes exactly one step – the Markov property drawn – and every reading hangs off its own state and nothing else. Nothing points back up: a reading reports on the pump, it never drives it. The twin's job is to maintain the belief over the hidden row using the observed one, and the recipe is the two steps already built: predict with Equation A.21, then correct with Bayes' rule, using the sensor's probabilities as the likelihood. A twin for a civil structure that carries its health state as a hidden variable and its sensor readings as the observed row is this figure with more states [175].

A.7.9 Worked example 3: one correction step

After two hours of silence the belief is $b_2 = \begin{bmatrix} 0.61 & 0.384 & 0.006 \end{bmatrix}$. The sensor now reports current. The likelihood of that reading in each state is $\begin{bmatrix} 0.05 & 0.95 & 0.3 \end{bmatrix}$. Bayes' rule, state by state, is prior times likelihood: Off, $0.61 \times 0.05 = 0.0305$; On, $0.384 \times 0.95 = 0.3648$; Faulted, $0.006 \times 0.3 = 0.0018$. The total is 0.3971, and dividing gives the posterior $\begin{bmatrix} 0.077 & 0.919 & 0.005 \end{bmatrix}$. One reading moved the belief from "probably idle" to "almost surely running", and the faulted entry is lower than before the reading, because a faulted pump draws current less often than a running one. Predict again with the table, correct again with the next reading, and you have a state estimator for a pump; do it with Gaussians and matrices instead of rows and lists, and you have a Kalman filter.

A.8 What a Solver and a Simulator Do

A.8.1 A solver runs the loop

A **simulator** is the program that runs a model forward in time. For a differential equation, the part of the simulator that does the running is the **solver**: it holds the state, chooses the step size, asks the equation for the rate, applies a step, and records the result. The loop of the From an Equation to a Loop section is a solver with the step chosen once, up front. A production solver chooses it as it goes, as the Better Solvers subsection described, and offers a choice of step formula; its interface is what you would design yourself – a function f that returns the rate given the state, an initial state, a time span, and a tolerance.

A.8.2 An event scheduler runs the discrete version

A discrete model – the pump chain, a conveyor, a message queue – has no rate to integrate. Nothing happens between events, so a solver that stepped through the quiet would waste every step. Its simulator is an **event scheduler**: a priority queue of pending events ordered by time, and a loop that pops the earliest, advances the clock to it, runs the handler, and pushes whatever new events the handler schedules.

```
queue = [(0.0, "start")]              # (time, event), earliest first
clock = 0.0
while queue not empty and clock < end:
    (t, event) = pop_earliest(queue)
    clock = t                         # time jumps; nothing happened between
    for (dt, next_event) in handle(event, state):
        push(queue, (clock + dt, next_event))
```

Time jumps from event to event, which is why a discrete simulation of a year of pump operation can finish in a millisecond, and why the same model in a fixed-step loop would take a step every second for a year. A discrete-event model is a model written for this loop; a conveyor or a print queue runs on it.

A.8.3 Worked example 1: the cost of a day of coffee

Cost of a run is **operations per step times number of steps**. One Euler step of the cooling rule is a subtraction, two multiplications and an addition: call it four operations. A day is 1440 minutes. At $h = 1$ minute that is 1440 steps and about 5,760 operations; at $h = 0.1$, 14,400 steps and 57,600 operations; at $h = 0.01$, 144,000 steps and 576,000 operations. A processor doing a billion simple operations a second finishes the finest of those in under a millisecond. For the coffee, step size is a question about accuracy alone; cost never enters. For the coupled coffee and mug, one step is a two-by-two times a vector – four multiplications and two additions – plus two additions for the step, so the cost roughly doubles and still does not matter. What changes the picture is dimension, in the third example.

A.8.4 Worked example 2: five events from the queue

Run the pump's scheduler for five events. The rules: a start command is due two hours after the pump last stopped; a running pump stops after one and a half hours; a failure is not scheduled here, to keep the trace short. Table A.1 lists what the loop does.

Table A.1: Five pops of the pump's event queue. The clock jumps to each event's time; nothing is computed between them.

Pop	Clock (h)	Event	State after	Pushed
1	0.0	start	Off	(2.0, switch_on)
2	2.0	switch_on	On	(3.5, switch_off)
3	3.5	switch_off	Off	(5.5, switch_on)
4	5.5	switch_on	On	(7.0, switch_off)
5	7.0	switch_off	Off	(9.0, switch_on)

Read Table A.1 down the clock column: seven hours of pump operation in five iterations, each a pop, a handler and a push. The queue never holds more than one event here; a conveyor with twenty parts in flight holds twenty, and the priority queue is what keeps the earliest at the front. The cost of a discrete run is operations per event times number of events, and the number of events is a property of the model's activity, not of the clock.

A.8.5 The real-time factor

One number decides whether a model can live inside a twin. The **real-time factor** is simulated time covered divided by the wall-clock time it took to cover it:

$$RTF = \frac{t_{\text{sim}}}{t_{\text{wall}}} \tag{A.24}$$

Both times in Equation A.24 are in the same unit, so the factor is a pure number. An RTF of one means the model keeps exact pace with the world. Below one, it falls behind: a twin running it can never catch up to the present and so cannot compare a prediction with the reading that has only now arrived. Above one, it runs ahead, and how far above one decides what the twin can do with it: at $RTF = 10$ it can run ten what-if scenarios in the time the world takes to run one; at $RTF = 1000$ it can run a day of scenarios during the coffee break. The distinction between a shadow and a twin lives here: a model whose RTF is below one can shadow nothing, and any use beyond monitoring needs it well above one.

A.8.6 Worked example 3: which model can live inside the twin

The coffee model at $h = 1$ minute took about 5,760 operations for a day. At a billion operations a second that is $t_{\text{wall}} = 5.76 \times 10^{-6}$ seconds for $t_{\text{sim}} = 86{,}400$ seconds, so $RTF = 86{,}400/(5.76 \times 10^{-6}) = 1.5 \times 10^{10}$ – fifteen billion times faster than the world. It can live inside any twin, run every what-if anyone asks for, and never be noticed.

Now the million-state tower. One step is 10^{12} multiplications if the table were dense; a real finite-element solver exploits the zeros and does perhaps 10^9 operations a step, and needs a step of one millisecond to resolve the tower's highest mode of interest. A day is 8.64×10^7 steps and about 8.64×10^{16} operations. At 10^{10} operations a second on a good workstation that is $t_{\text{wall}} = 8.64 \times 10^6$ seconds, a hundred days, for one simulated day: $RTF = 0.01$. This model cannot live inside the twin, for any purpose. The answer is not a faster computer – a hundred-fold shortfall is not closed that way – but a different model: a reduced-order model that keeps the few modes the Many Quantities section described, with n in the tens, an RTF in the thousands, and an error that then has to be accounted for. The real-time factor is the number that forces that decision, and the reason a twin holds more than one kind of model at all.

A.9 Notation Used in This Book

The book reserves a small set of letters and uses them the same way in every chapter. Table A.2 collects them, with the way each is read aloud, so that a symbol met in a chapter can be looked up here in one place.

Table A.2: The letters the book reserves, how each is read aloud, and the section of this appendix that builds it.

Symbol	Read as	Meaning	Built in
T, $T(t)$	"T", "T at time t"	a quantity; the same quantity at a named moment	Quantities and Rates
dT/dt	"the rate of T"	how fast T is changing now; the derivative	Quantities and Rates
θ	"theta"	the parameters of a rule, spoken of together	Quantities and Rates
h	"the step"	the time one step of the loop advances	From an Equation to a Loop
x_k	"x at step k"	a quantity after k steps; a subscript always counts steps, never time	Quantities, Rates and Equations
x	"the state"	the record of everything the model must remember; dimension n	Many Quantities
u	"the input"	what is done to the system from outside: a command, a load	Many Quantities
z	"the output" or "the reading"	what a sensor reports	Many Quantities
w_k, v_k	"process noise", "measurement noise"	what the rule leaves out; what the sensor gets wrong	The Markov Chain
f, g	"the step rule", "the output rule"	next state from present state and input; reading from state	Many Quantities

Symbol	Read as	Meaning	Built in
A, A^{T}, A^{-1}, I	"A", "A transpose", "A inverse", "the identity"	a table of coefficients; rows and columns swapped; the table that undoes it; the table that changes nothing	Many Quantities
B, C, F, G	"B", "C", "F", "G"	the input table; the output table; the discrete step table; the discrete input table	Many Quantities
λ, v, ω	"lambda", "v", "omega"	a mode's rate; its shape; a natural frequency in radians per second	Many Quantities
$p(x)$	"the density of x"	a curve whose areas are probabilities	Probability
μ, σ, σ^2	"mu", "sigma", "sigma squared"	mean; standard deviation, in the quantity's units; variance	Probability
P, Q, R	"P", "Q", "R"	covariance of the state; of the process noise; of the measurement noise	Probability
$\mathcal{N}(\mu, \sigma^2)$	"normal with mean mu and variance sigma squared"	the Gaussian density	Probability
$p(x \mid z)$	"the density of x given z"	conditional density; the bar is "given"	Probability
Π, b_k	"the transition matrix", "the belief at step k"	a row-stochastic table; a row of probabilities over the states	The Markov Chain
$\hat{x}$	"x hat"	an estimate of x; a hat always marks an estimate, never a true value	Probability
RTF	"the real-time factor"	simulated time covered per unit of wall-clock time	Solver and Simulator

Two of those letters clash with the symbols the book's watering-rig case study uses, and the book keeps both rather than renaming either. Table A.3 lists the rig's symbols and says where

each clash is resolved.

Table A.3: The watering rig's symbols beside the reserved letters they share, and how a reader tells them apart.

Rig symbol	Meaning for the watering rig	Clash, and how it is read
m, m_{t}	soil moisture; target moisture	none; the appendix uses m for a matrix's row count only inside the Many Quantities section
V, $V_{\max}$	dose asked for; the largest dose	none
T_{r}, T_{c}	the sensor's reading period, seconds; the twin's check interval, hours	none; both are step sizes in this appendix's sense
k	the controller's gain, millilitres per point of deficit	the appendix's cooling constant is also k, local to the coffee example; in a subscript, k always counts steps. The context – a rig, a coffee, a subscript – says which
d, g	drying rate; dose response	g is also the book's output rule in the state-space model; the rig's g carries units of points per millilitre and the output rule carries none
r, $\hat{m}$	the residual; the twin's predicted moisture	none; $\hat{m}$ is the hat convention above, applied to the rig

The mapped-mathematics convention, restated as a table, is Table A.4: what is typed in the book's source and what it becomes on the page.

Table A.4: How the book's plain-text source spells mathematics, and what the rendered page shows for each spelling.

Typed in the source as	Becomes on the page	Example
an asterisk between two symbols	multiplication, usually shown as a thin space or a times sign	$a\,b$
a forward slash	division, as a fraction where the line is displayed	a/b
a caret	a power	a^2
an underscore	a subscript	x_k
a comma between two relations, on one line	two equations set side by side	$y_1 = T_1 - T_{\mathrm{a}}, y_2 = T_2 - T_{\mathrm{a}}$

Typed in the source as	Becomes on the page	Example
semicolons between rows inside brackets	a matrix or a column vector	$\left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]$
the word `integral` with limits after it	an integral sign with those limits	$\int_{\alpha}^{\beta} p(x)\,dx$
a hat, transpose or inverse spelled as a suffix	the hat, the superscript T, the superscript minus one	$\hat{x}$, A^{T}, A^{-1}

Three lines read aloud, as worked examples of the table. First, $x_{k+1} = f(x_k, u_k, \theta) + w_k$ is "x at step k plus one equals f of x at step k, u at step k and theta, plus the process noise at step k". Second, $P(\text{hot} \mid \text{alarm}) = 0.32$ is "the probability of hot given alarm is 0.32". Third, $b_{k+1} = b_k\,\Pi$ is "the belief at step k plus one equals the belief at step k times the transition matrix". A reader who can produce those three sentences from the symbols can read every equation in the book.

A.10 Where Each Section Is Used

Table A.5 runs from each section of this appendix to the chapters and sections that lean on it. This is the one place the appendix names a chapter. Chapters not yet written are named by their role in the book and marked as planned.

Table A.5: From each section of this appendix to the chapters and sections that use it. Planned chapters are named by role.

Appendix section	Leaned on by
Quantities, Rates and Equations	Chapter 2, Watering Strategies and Plant Digital Twin (the drying rate d and the dose response g as rate rules; the controller as a difference equation); Chapter 3, Model, Simulation and Simulator; the models chapter, the lumped first-principles model type (planned)
The Integration	Chapter 2, Plant Digital Twin (the predicted moisture between checks as an integral of the drying rate); the models chapter, where a closed form exists and where it does not (planned)
From an Equation to a Loop	Chapter 5, Sampling and Time (sampled time, the sampling interval as the step); the models chapter, the simulators half and its solver choices (planned); the state and estimator chapter, the prediction step (planned)

Appendix section	Leaned on by
Many Quantities at Once	Chapter 4, Digital Twin Configuration (the state as a record); Chapter 5, Sampling (a mode's natural frequency sets the interval); the models chapter, the high-fidelity and reduced-order model types and stiffness (planned); the state and estimator chapter, the state-space form throughout (planned)
Probability and Distributions	Chapter 4, Digital Twin Configuration (uncertainty as a field of state); Chapter 5, Sensors and Quality (accuracy as a standard deviation); the state and estimator chapter, the correction step and Bayes' rule (planned); the credibility chapter, stated uncertainty (planned)
State Machines with Probabilities	Chapter 4, Digital Twin Configuration and the `as_of` field (the belief and its leak); the models chapter, the discrete-state and probabilistic model types (planned); the state and estimator chapter, the hidden Markov model (planned)
What a Solver and a Simulator Do	Chapter 1, Six Value Patterns (what each pattern demands of the real-time factor); Chapter 3, Model, Simulation and Simulator; the models chapter, the simulators half and its real-time factor (planned); the standards chapter, co-simulation (planned)
Notation Used in This Book	every chapter from Chapter 2 onward

Read Table A.5 down the right-hand column and one chapter appears against every row but the last: the state and estimator chapter is the one that needs all of this appendix, and a reader headed there should read it whole. Every other chapter needs one or two sections, which is why the appendix is arranged so that they can be read alone.

What this appendix established, against its objectives. A differential equation and an initial condition fix a trajectory, the rate at any state is one arithmetic step from the equation, and a difference equation is the same idea with the step built in (objective 1). The integral form says how much has accumulated, a closed form exists for the cooling law and few others, and Euler's step is one rectangle of that integral, with Heun's trapezium and the implicit step as the two improvements worth knowing (objective 2). A matrix times a vector is two loops, a transpose swaps rows and columns, a two-by-two inverts by a formula that fails when its determinant is zero, a system of equations is one matrix rule, the state-space form adds input and output tables, and a mode is a vector the matrix only stretches (objective 3). A probability is a ratio of counts, a conditional probability restricts the count, Bayes' rule turns a conditional around, independence is a test, and a density's areas are probabilities, with the uniform and the Gaussian as the two shapes that matter (objective 4). A Markov chain is a state machine with a probability on every arrow, a belief predicts by one multiplication and corrects by Bayes' rule, and the Markov property is what lets a state be one row (objective 5). A solver runs the loop, a scheduler runs

the queue, cost is operations per step times steps, and the real-time factor decides which model a twin can carry (objective 6).

A.11 Exercises

Each exercise names the learning objective it serves. Solutions follow, or a hint where the exercise is open.

1. *(Objective 1)* A vacuum flask has $k = 0.02$ per minute. Its tea is at 90 degrees in a room at 20. State the rate of change of the tea's temperature, name the state, the parameters and the initial condition, and say where the trajectory ends.
2. *(Objective 1)* A tank with a flat bottom of area 2 square metres is filled at 0.5 cubic metres a minute and drained at 0.3. Write the differential equation for the water level L and compute the rate. Which of the numbers are parameters, what is the state, and what is the difference equation for a level sensor read once a minute?
3. *(Objective 2)* Using the closed form, find when the paper cup ($k = 0.2$, $T_\mathrm{a} = 20$, $T(0) = 80$) reaches 30 degrees, and its time constant.
4. *(Objective 2)* Step the coffee ($k = 0.1$, $T_\mathrm{a} = 20$, $T_0 = 80$) forward two Euler steps with $h = 2$ minutes, then one Heun step with $h = 2$. Compare both with the closed-form 69.12 at minute two.
5. *(Objective 2)* A system has $k = 0.5$ per minute. What is the largest step at which Euler's loop stays stable? A probe in the same model has $k = 100$. What step does the pair need, what word describes the pair, and which method from the Better Solvers subsection removes the constraint?
6. *(Objective 3)* $A = \left[\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right]$ and $x = \left[\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right]$. Compute $A\,x$, A^T and A^{-1}, and check the inverse by applying it to $A\,x$.
7. *(Objective 3)* Write the discrete state-space table $F = I + hA$ for the coffee and mug at $h = 0.5$ minutes, and apply it to $\left[\begin{smallmatrix} 60 \\ 0 \end{smallmatrix}\right]$. Why is the result not half of the $h = 1$ result?
8. *(Objective 3)* From the two modes of the coffee and mug, state the largest stable Euler step, and say which mode sets it.
9. *(Objective 4)* A reading is rounded to the nearest four units, so the true value is uniform on an interval of width four. Compute the probability it lies in the middle half of the interval, and its standard deviation.
10. *(Objective 4)* Repeat the bearing example with a better sensor: $P(\text{alarm} \mid \text{hot}) = 0.99$ and $P(\text{alarm} \mid \text{normal}) = 0.01$, prior 0.05. What is the posterior after one alarm, and why is it so much higher than 0.32?
11. *(Objective 4)* A thermometer has $\sigma = 0.5$ degrees. Out of 300 readings, about how many would you expect to be more than one degree off, and more than one and a half degrees off? A second thermometer of the same kind is added; state the assumption under which the probability that both are more than a degree off on the same reading is the product of the two, and one way the assumption could fail.
12. *(Objective 5)* From $b_2 = \left[\begin{smallmatrix} 0.61 & 0.384 & 0.006 \end{smallmatrix}\right]$, compute b_3 by one more multiplication with the pump's table, and check that it sums to one. Then, instead, suppose the sensor at hour two reports *no* current; compute the posterior.
13. *(Objective 5)* The pump's failure probability is found to grow with the number of hours it has run since its last repair. The chain as drawn cannot express that. What fails, and what is the repair?
14. *(Objective 6)* A reduced-order tower model has $n = 20$, costs about 2,000 operations a step, and needs a step of 10 milliseconds. Compute its real-time factor on a machine doing 10^{10} operations a second, and say what a twin can do with it.

15. *(Objective 2, Objective 6)* Take the implicit Euler step for the coffee with $h = 10$ and with $h = 25$, compare each with the closed form and with the explicit step, and say what the implicit step buys a twin that must run a stiff model in real time.

A.11.1 Solutions and hints

Exercise 1 (solution). Rate $-0.02 \times (90 - 20) = -1.4$ degrees per minute. State T; parameters $k = 0.02$ and $T_a = 20$; initial condition $T(0) = 90$. The trajectory ends at the room temperature, 20, where the rate is zero.

Exercise 2 (solution). The level rises by the net inflow spread over the bottom: $dL/dt = (q_{in} - q_{out})/A_{tank}$, so $dL/dt = (0.5 - 0.3)/2 = 0.1$ metres per minute. The state is L; the three numbers are parameters – the area and the two flows – and the initial condition, which the exercise does not give, is whatever the level is when the clock starts. The difference equation is $L_{k+1} = L_k + 0.1$ with one step per minute, exact because the rate does not depend on L; the level is a straight line, like the drying soil.

Exercise 3 (solution). The excess must fall from 60 to 10, so $\exp(-0.2\,t) = 1/6$ and $t = \ln(6)/0.2 = 8.96$ minutes, half the ceramic cup's 17.9 because k doubled. The time constant is $1/k = 5$ minutes.

Exercise 4 (solution). Euler, step one: rate -6, so $T_1 = 80 - 12 = 68$. Step two: rate $-0.1 \times 48 = -4.8$, so $T_2 = 68 - 9.6 = 58.4$; that is minute four, so compare instead after one Euler step of $h = 2$ at minute two: 68 against 69.12, an error of 1.12. Heun with $h = 2$: start rate -6, guess 68, rate there -4.8, $T_1 = 80 + 1 \times (-6 - 4.8) = 69.2$, against 69.12, an error of 0.08 – fourteen times smaller for one extra rate evaluation.

Exercise 5 (hint). Stability needs $h < 2/k$, so four minutes for the system and about 0.02 minutes – a little over a second – for the probe. The pair needs the probe's step, everywhere, for the whole run, though the probe settles in seconds; that is stiffness. The implicit Euler step is stable at any step size, so the step can be chosen for the accuracy of the slow part alone.

Exercise 6 (solution). $A\,x = \left[\begin{smallmatrix} 1+2 \\ 3+4 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 3 \\ 7 \end{smallmatrix}\right]$. $A^{\mathsf{T}} = \left[\begin{smallmatrix} 1 & 3 \\ 2 & 4 \end{smallmatrix}\right]$. Determinant $1 \times 4 - 2 \times 3 = -2$, so $A^{-1} = \left[\begin{smallmatrix} -2 & 1 \\ 1.5 & -0.5 \end{smallmatrix}\right]$. Applying it to $\left[\begin{smallmatrix} 3 \\ 7 \end{smallmatrix}\right]$: row one $-2 \times 3 + 1 \times 7 = 1$, row two $1.5 \times 3 - 0.5 \times 7 = 1$. The original $\left[\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right]$ comes back.

Exercise 7 (solution). $F = \left[\begin{smallmatrix} 0.95 & 0.05 \\ 0.15 & 0.75 \end{smallmatrix}\right]$. Applied to $\left[\begin{smallmatrix} 60 \\ 0 \end{smallmatrix}\right]$: $\left[\begin{smallmatrix} 57 \\ 9 \end{smallmatrix}\right]$. It is not half of $\left[\begin{smallmatrix} 54 \\ 18 \end{smallmatrix}\right]$'s *change* because the identity part of F does not scale with h: the change from $\left[\begin{smallmatrix} 60 \\ 0 \end{smallmatrix}\right]$ is $\left[\begin{smallmatrix} -3 \\ 9 \end{smallmatrix}\right]$, which *is* half of the $h = 1$ change $\left[\begin{smallmatrix} -6 \\ 18 \end{smallmatrix}\right]$. Halving the step halves each step's movement, not the state.

Exercise 8 (solution). The fast mode has $\lambda_2 = -0.5646$ per minute, so Euler is stable only for $h < 2/0.5646 = 3.54$ minutes. The slow mode, with $\lambda_1 = -0.0354$, would allow h up to 56 minutes on its own; the fast mode sets the step, which is the stiffness the section described.

Exercise 9 (solution). The density is 1/4 on the interval. The middle half has width two, so the probability is $2/4 = 0.5$. The variance of a uniform density of width W is $W^2/12$, here $16/12 = 1.33$, so $\sigma = 1.15$ units – again about 0.29 of the width.

Exercise 10 (solution). Hot: $0.99 \times 0.05 = 0.0495$. Normal: $0.01 \times 0.95 = 0.0095$. Total 0.059. Posterior $0.0495/0.059 = 0.84$. It is higher because the false-alarm rate fell from 10 per cent to 1 per cent: the normal bearings, which are nineteen times as common, now contribute nineteen times fewer false alarms, so the alarm carries more evidence. The prior did not change; the likelihood did.

Exercise 11 (solution). One degree is two standard deviations, so about 5 per cent of readings, fifteen of the 300, are more than a degree off; one and a half degrees is three standard deviations, so about 0.3 per cent, one reading. The product $0.05 \times 0.05 = 0.0025$ holds only if the two thermometers' errors are independent; a shared cold draught across both probes, or a shared reference voltage, makes both err the same way at once and the joint probability larger than the product.

Exercise 12 (solution). Off: $0.61 \times 0.7 + 0.384 \times 0.4 + 0.006 \times 0.1 = 0.5812$. On: $0.61 \times 0.3 + 0.384 \times 0.58 + 0 = 0.4057$. Faulted: $0.384 \times 0.02 + 0.006 \times 0.9 = 0.0131$. Sum 1.000. The largest entry is now 0.58, the leak continuing. For the reading of no current the likelihoods are one minus the current likelihoods, $\begin{bmatrix} 0.95 & 0.05 & 0.7 \end{bmatrix}$: Off 0.5795, On 0.0192, Faulted 0.0042, total 0.6029, posterior $\begin{bmatrix} 0.961 & 0.032 & 0.007 \end{bmatrix}$. The reading confirmed the belief rather than overturning it, and the faulted entry barely moved, because "no current" is consistent with both Off and Faulted.

Exercise 13 (hint). The Markov property fails: the next state depends on how long the pump has run, which is not in the state. The repair is to put it there – states such as "On, under 100 hours" and "On, over 100 hours", or a run-hours quantity beside the discrete state – and not to abandon the chain.

Exercise 14 (solution). A day is 8,640,000 steps of 10 milliseconds; at 2,000 operations each, 1.73×10^{10} operations, which is 1.7 seconds at 10^{10} a second. $RTF = 86{,}400/1.7 = 5 \times 10^4$. The twin can run fifty thousand simulated days in one real day, or a hundred what-if scenarios for the coming hour in seven seconds; prediction and decision support are open to it, at the cost of whatever accuracy the reduction gave up.

Exercise 15 (solution). With $h = 10$: $T_1 = (80 + 1 \times 20)/(1 + 1) = 50$, against a closed form of 42.07 and an explicit step of 20. With $h = 25$: $(80 + 2.5 \times 20)/3.5 = 37.1$, against 22.99 at minute 25 and an explicit -70. The implicit step is less accurate than the exact curve at these long steps but never unstable, so a stiff model can be stepped at the rate its slow part needs – and its real-time factor rises by the ratio of the two steps – instead of at the rate its fastest transient demands.

A.12 Further Reading

Grouped by section. The proofs this appendix omits are in the standard first-year texts of each subject, and any of them will do; the entries below are sources this book's own chapters draw on, and where in a twin each idea appears.

- **Equations, integration and the loop.** A foundations text for modelling cyber-physical systems derives Euler's error, introduces the Runge-Kutta family and motivates adaptive step size [166]. A coupled building-simulation system describes in practical terms what an adaptive-step solver does with a stiff mix of continuous and switching dynamics [169].
- **Vectors, systems and state space.** A twin of a laboratory incubator walks from a continuous state-space model through discretisation to the discrete system a state estimator runs on [190]; a physics-based twin of a floating wind turbine runs a Kalman filter over a linear structural model to estimate section loads along the tower [164]; and a structural-monitoring paper explains natural frequencies and mode shapes as the quantities operational modal analysis extracts from vibration data [191].
- **Probability and the Markov chain.** The formal picture of a twin and its asset as coupled dynamical systems carried by a probabilistic graphical model is set out for an unmanned aircraft [171], and applied to a civil structure with a dynamic Bayesian network carrying the observations-to-decisions flow [175]. Both are the hidden Markov model of this appendix at full scale.

Bibliography

[1] T. Bergs, S. Gierlings, T. Auerbach, A. Klink, D. Schraknepper, and T. Augspurger, "The concept of digital twin and digital shadow in manufacturing," *Procedia CIRP*, vol. 101, pp. 81–84, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827121006612

[2] R. Bhandal, "Conceptualising the Application of Digital Twins in Supply Chain Management: A Path Towards Supply Chain Resilience," in *Digital Twin: Fundamentals and Applications*, S. Sabri, K. Alexandridis, and N. Lee, Eds. Cham: Springer Nature Switzerland, 2024, pp. 173–189. [Online]. Available: https://doi.org/10.1007/978-3-031-67778-6_8

[3] "Case studies of digitalization for creating digital twins." [Online]. Available: https://www.change2twin.eu/about/deliverables/

[4] T. Milligan, "Infrastructure Lifecycle: A Case for Change."

[5] M. Grieves and J. Vickers, "Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems," in *Transdisciplinary Perspectives on Complex Systems: New Findings and Approaches*, F.-J. Kahlen, S. Flumerfelt, and A. Alves, Eds. Cham: Springer International Publishing, 2017, pp. 85–113. [Online]. Available: https://doi.org/10.1007/978-3-319-38756-7_4

[6] O. J. Pinon Fischer, S. Sabri, and Y. Chen, "Fundamentals of Digital Twins, Modeling Approaches, and Governance," in *Digital Twin: Fundamentals and Applications*, S. Sabri, K. Alexandridis, and N. Lee, Eds. Cham: Springer Nature Switzerland, 2024, pp. 13–29. [Online]. Available: https://doi.org/10.1007/978-3-031-67778-6_2

[7] "Digital Twin Driven Smart Manufacturing," in *Digital Twin Driven Smart Manufacturing*, F. Tao, M. Zhang, and A. Y. C. Nee, Eds. Academic Press, Jan. 2019, pp. i–iii. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128176306000138

[8] "Digital Engineering Effectiveness," May 2022. [Online]. Available: https://insights.sei.cmu.edu/library/digital-engineering-effectiveness/

[9] K. Y. H. Lim, P. Zheng, and C.-H. Chen, "A state-of-the-art survey of Digital Twin: techniques, engineering product lifecycle management and business innovation perspectives," *Journal of Intelligent Manufacturing*, vol. 31, no. 6, pp. 1313–1337, Aug. 2020. [Online]. Available: https://doi.org/10.1007/s10845-019-01512-w

[10] S. Mihai, M. Yaqoob, D. V. Hung, W. Davis, P. Towakel, M. Raza, M. Karamanoglu, B. Barn, D. Shetve, R. V. Prasad, H. Venkataraman, R. Trestian, and H. X. Nguyen, "Digital Twins: A Survey on Enabling Technologies, Challenges, Trends and Future Prospects," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 4, pp. 2255–2291,

2022, conference Name: IEEE Communications Surveys & Tutorials. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9899718

[11] A. Thelen, X. Zhang, O. Fink, Y. Lu, S. Ghosh, B. D. Youn, M. D. Todd, S. Mahadevan, C. Hu, and Z. Hu, "A comprehensive review of digital twin—part 2: roles of uncertainty quantification and optimization, a battery digital twin, and perspectives," *Structural and Multidisciplinary Optimization*, vol. 66, no. 1, p. 1, Dec. 2022. [Online]. Available: https://doi.org/10.1007/s00158-022-03410-x

[12] G. Lugaresi and H. Vangheluwe, *From Digital Twins to Twinning Systems*, Dec. 2025.

[13] T. R. Wanasinghe, L. Wroblewski, B. K. Petersen, R. G. Gosine, L. A. James, O. De Silva, G. K. I. Mann, and P. J. Warrian, "Digital Twin for the Oil and Gas Industry: Overview, Research Trends, Opportunities, and Challenges," *IEEE Access*, vol. 8, pp. 104 175–104 197, 2020, conference Name: IEEE Access. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9104682

[14] E. Katsoulakis, Q. Wang, H. Wu, L. Shahriyari, R. Fletcher, J. Liu, L. Achenie, H. Liu, P. Jackson, Y. Xiao, T. Syeda-Mahmood, R. Tuli, and J. Deng, "Digital twins for health: a scoping review," *npj Digital Medicine*, vol. 7, no. 1, pp. 1–11, Mar. 2024. [Online]. Available: https://www.nature.com/articles/s41746-024-01073-0

[15] D. McKee and D. Dokter, "DISCS: An Approach for Accelerating the Development of Digital Twins for Smart Cities," in *Digital Twin: Fundamentals and Applications*, S. Sabri, K. Alexandridis, and N. Lee, Eds. Cham: Springer Nature Switzerland, 2024, pp. 31–58. [Online]. Available: https://doi.org/10.1007/978-3-031-67778-6_3

[16] D. Parle, G. Sharma, N. Anand, N. Padgaonkar, D. Stoddart, and D. Malley, "A Comparative Analysis for Harnessing Digital Twin Platforms for Net-Zero Manufacturing," *IEEE Access*, vol. PP, Aug. 2024.

[17] R. Saracco, "Digital Twins: Bridging Physical Space and Cyberspace," *Computer*, vol. 52, no. 12, pp. 58–64, Dec. 2019, conference Name: Computer. [Online]. Available: https://ieeexplore.ieee.org/document/8909940

[18] P. van Schalkwyk and D. Isaacs, "Achieving Scale Through Composable and Lean Digital Twins," in *The Digital Twin*, N. Crespi, A. T. Drobot, and R. Minerva, Eds. Cham: Springer International Publishing, 2023, pp. 153–180. [Online]. Available: https://doi.org/10.1007/978-3-031-21343-4_6

[19] "The Industrial Internet Reference Architecture." [Online]. Available: https://www.iiconsortium.org/iira/

[20] D. Hartmann and H. Van der Auweraer, *The Executable Digital Twin: merging the digital and the physics worlds*, Oct. 2022.

[21] M.-E. Iliuţă, M.-A. Moisescu, E. Pop, A.-D. Ionita, S.-I. Caramihai, and T.-C. Mitulescu, "Digital Twin—A Review of the Evolution from Concept to Technology and Its Analytical Perspectives on Applications in Various Fields," *Applied Sciences*, vol. 14, no. 13, p. 5454, Jan. 2024, number: 13. [Online]. Available: https://www.mdpi.com/2076-3417/14/13/5454

[22] F. Stadtmann and A. Rasheed, "Diagnostic Digital Twin for Anomaly Detection in Floating Offshore Wind Energy," Jun. 2024, arXiv:2406.02775 [cs]. [Online]. Available: http://arxiv.org/abs/2406.02775

[23] H. Zhang, Z. Wang, S. Zhang, L. Qiu, Y. Wang, F. Xiang, Z. Pan, L. Zhu, and J. Tan, "Digital-Triplet: a new three entities digital-twin paradigm for equipment fault diagnosis," *Journal of Intelligent Manufacturing*, Aug. 2024. [Online]. Available: https://link.springer.com/10.1007/s10845-024-02471-7

[24] E. Altamiranda, "A System of Systems Foundation for Digital Asset Lifecycle Management," in *Digital Twin: Fundamentals and Applications*, S. Sabri, K. Alexandridis, and N. Lee, Eds. Cham: Springer Nature Switzerland, 2024, pp. 59–87. [Online]. Available: https://doi.org/10.1007/978-3-031-67778-6_4

[25] R. van Dinter, B. Tekinerdogan, and C. Catal, "Predictive maintenance using digital twins: A systematic literature review," *Information and Software Technology*, vol. 151, p. 107008, Nov. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922001331

[26] D. Zhong, Z. Xia, Y. Zhu, and J. Duan, "Overview of predictive maintenance based on digital twin technology," *Heliyon*, vol. 9, no. 4, p. e14534, Apr. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405844023017413

[27] A. Thelen, X. Zhang, O. Fink, Y. Lu, S. Ghosh, B. D. Youn, M. D. Todd, S. Mahadevan, C. Hu, and Z. Hu, "A comprehensive review of digital twin — part 1: modeling and twinning enabling technologies," *Structural and Multidisciplinary Optimization*, vol. 65, no. 12, p. 354, Nov. 2022. [Online]. Available: https://doi.org/10.1007/s00158-022-03425-4

[28] C. Gomes, M. H. Kristensen, M. S. Andersen, P. Talasila, H. Feng, T. Wright, and P. G. Larsen, "Digital Twin Tutorial: The Incubator Case Study," in *Engineering Trustworthy Software Systems: 6th International School, SETSS 2024, Chongqing, China, April 14–21, 2024, Tutorial Lectures*, J. P. Bowen, C. Gomes, and Z. Liu, Eds. Singapore: Springer Nature, 2025, pp. 68–101. [Online]. Available: https://doi.org/10.1007/978-981-96-4656-2_3

[29] Y. Jiang, S. Yin, K. Li, H. Luo, and O. Kaynak, "Industrial applications of digital twins," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 379, no. 2207, p. 20200360, Aug. 2021. [Online]. Available: https://royalsocietypublishing.org/doi/full/10.1098/rsta.2020.0360

[30] M. Frasheri, T. Böttjer, P. G. Larsen, L. Esterle, and C. Gomes, "Advanced Digital Twin Services," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds. Cham: Springer International Publishing, 2024, pp. 209–222. [Online]. Available: https://doi.org/10.1007/978-3-031-66719-0_10

[31] A. Barbie and W. Hasselbring, "Toward Reproducibility of Digital Twin Research: Exemplified with the PiCar-X," Aug. 2024, arXiv:2408.13866 [cs]. [Online]. Available: http://arxiv.org/abs/2408.13866

[32] F. Zampetti, D. Tamburri, S. Panichella, A. Panichella, G. Canfora, and M. Di Penta, "Continuous Integration and Delivery Practices for Cyber-Physical Systems: An Interview-Based Study," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 3, pp. 73:1–73:44, Apr. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3571854

[33] "Change2Twin." [Online]. Available: https://www.change2twin.eu/about/download/

[34] T. Alskaif, Ö. Babur, F. Bordeleau, L. Cleophas, B. Combemale, J. Denil, Ø. Haugen, J. Michael, P. Nguyen, T. Seceleanu, M. van den Brand, and H. Vangheluwe, "Evolution

at the Core of Digital Twin Engineering." Grand Rapids, USA: IEEE, Jul. 2025. [Online]. Available: https://epub.uni-regensburg.de/77656/

[35] M. Perno, L. Hvam, and A. Haug, "Implementation of digital twins in the process industry: A systematic literature review of enablers and barriers," *Computers in Industry*, vol. 134, p. 103558, Jan. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166361521001652

[36] S. A. Niederer, M. S. Sacks, M. Girolami, and K. Willcox, "Scaling digital twins from the artisanal to the industrial," *Nature Computational Science*, vol. 1, no. 5, pp. 313–320, May 2021. [Online]. Available: https://www.nature.com/articles/s43588-021-00072-5

[37] P. Zech, C. Nardin, S. Ristov, M. Flora, and R. Breu, "Digital-Twins-as-a-Service in Construction Engineering," in *2024 IEEE 20th International Conference on Automation Science and Engineering (CASE)*, Aug. 2024, pp. 3004–3010. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10711409

[38] E. Ferko, A. Bucaioni, P. Pelliccione, and M. Behnam, "Standardisation in Digital Twin Architectures in Manufacturing," in *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, Mar. 2023, pp. 70–81. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10092613

[39] "Digital Twin: Definition & Value – An AIAA and AIA Position Paper." [Online]. Available: https://www.aia-aerospace.org/publications/digital-twin-definition-value-an-aiaa-and-aia-position-paper/

[40] H. Pezeshki, H. Adeli, D. Pavlou, and S. C. Siriwardane, "State of the art in structural health monitoring of offshore and marine structures," *Proceedings of the Institution of Civil Engineers - Maritime Engineering*, vol. 176, no. 2, pp. 89–108, Apr. 2023. [Online]. Available: https://www.icevirtuallibrary.com/doi/full/10.1680/jmaen.2022.027

[41] Q. Qi, F. Tao, T. Hu, N. Anwer, A. Liu, Y. Wei, L. Wang, and A. Y. C. Nee, "Enabling technologies and tools for digital twin," *Journal of Manufacturing Systems*, vol. 58, pp. 3–21, Jan. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S027861251930086X

[42] R. van Dinter, B. Tekinerdogan, and C. Catal, "Reference architecture for digital twin-based predictive maintenance systems," *Computers & Industrial Engineering*, vol. 177, p. 109099, Mar. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360835223001237

[43] P. Singh, N. Nidhi, J. Karpavice, M. Beliatis, and M. Presser, *Digital Dataspace and Business Ecosystem Growth for Industrial Roll-to-Roll Label Printing Manufacturing: A Case Study*, Sep. 2023.

[44] M. Viceconti, M. De Vos, S. Mellone, and L. Geris, "Position Paper From the Digital Twins in Healthcare to the Virtual Human Twin: A Moon-Shot Project for Digital Health Research," *IEEE Journal of Biomedical and Health Informatics*, vol. 28, no. 1, pp. 491–501, Jan. 2024, conference Name: IEEE Journal of Biomedical and Health Informatics. [Online]. Available: https://ieeexplore.ieee.org/document/10278402

[45] P. G. Larsen, J. Fitzgerald, and C. Gomes, "Engineering Digital Twins for Cyber-Physical Systems," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds. Cham: Springer International Publishing, 2024, pp. 3–17. [Online]. Available: https://doi.org/10.1007/978-3-031-66719-0_1

[46] B. J. Oakes, H. Zhang, L. I. Hatledal, H. Feng, M. Frasheri, M. Sandberg, S. Gil, and C. Gomes, "Case Studies in Digital Twins," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds. Cham: Springer International Publishing, 2024, pp. 257–310. [Online]. Available: https://doi.org/10.1007/978-3-031-66719-0_12

[47] J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds., *The Engineering of Digital Twins*. Cham: Springer International Publishing, 2024. [Online]. Available: https://link.springer.com/10.1007/978-3-031-66719-0

[48] Z. Ali, R. Biglari, J. Denil, J. Mertens, M. Poursoltan, and M. K. Traoré, "From modeling and simulation to Digital Twin: evolution or revolution?" *SIMULATION*, vol. 100, no. 7, pp. 751–769, Jul. 2024. [Online]. Available: https://journals.sagepub.com/doi/10.1177/00375497241234680

[49] A. Hakiri, A. Gokhale, S. B. Yahia, and N. Mellouli, "A comprehensive survey on digital twin for future networks and emerging Internet of Things industry," *Computer Networks*, vol. 244, p. 110350, May 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128624001828

[50] P. G. Larsen, H. D. Macedo, J. Fitzgerald, H. Pfeifer, M. Benedikt, S. Tonetta, A. Marguglio, G. Veneziano, L. Sutton, S. Gusmeroli, and G. Suciu, "HUBCAP: A Novel Collaborative Approach to Model-Based Design of Cyber-Physical Systems," in *Simulation and Modeling Methodologies, Technologies and Applications*, M. S. Obaidat, T. Oren, and F. D. Rango, Eds. Cham: Springer International Publishing, 2022, pp. 90–110.

[51] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, "Digital Twin in manufacturing: A categorical literature review and classification," *Ifac-PapersOnline*, vol. 51, no. 11, pp. 1016–1022, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896318316021

[52] L. F. Rivera, M. Jiménez, N. M. Villegas, G. Tamura, and H. A. Müller, "The forging of autonomic and cooperating digital twins," *IEEE Internet Computing*, vol. 26, no. 5, pp. 41–49, 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9325563/

[53] E. Kamburjan, R. Sieve, C. Prabhu Baramashetru, M. Amato, G. Barmina, E. Occhipinti, and E. Broch Johnsen, "GreenhouseDT: An Exemplar for Digital Twins," in *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '24. New York, NY, USA: Association for Computing Machinery, Jun. 2024, pp. 175–181. [Online]. Available: https://dl.acm.org/doi/10.1145/3643915.3644108

[54] E. Kamburjan, N. Bencomo, S. L. Tapia Tarifa, and E. B. Johnsen, "Declarative Lifecycle Management in Digital Twins," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*. Linz Austria: ACM, Sep. 2024, pp. 353–363. [Online]. Available: https://dl.acm.org/doi/10.1145/3652620.3688248

[55] S. Infante, C. Martín, J. Robles, B. Rubio, M. Díaz, R. González Perea, P. Montesinos, and E. Poyato, "Integrating FMI and ML/AI models on the open-source digital twin framework OpenTwins," *Software Practice and Experience*, Mar. 2024.

[56] B. Tekinerdogan and C. Verdouw, "Systems Architecture Design Pattern Catalog for Developing Digital Twins," *Sensors*, vol. 20, no. 18, p. 5103, Jan. 2020, number: 18. [Online]. Available: https://www.mdpi.com/1424-8220/20/18/5103

[57] N. Zhang, R. Bahsoon, N. Tziritas, and G. Theodoropoulos, "Knowledge Equivalence in Digital Twins of Intelligent Systems," *ACM Transactions on Modeling and Computer Simulation*, vol. 34, no. 1, pp. 1–37, Jan. 2024, arXiv:2204.07481 [cs, eess]. [Online]. Available: http://arxiv.org/abs/2204.07481

[58] K. M. Alam and A. El Saddik, "C2PS: A digital twin architecture reference model for the cloud-based cyber-physical systems," *IEEE access*, vol. 5, pp. 2050–2062, 2017. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7829368/

[59] E. Ferko, A. Bucaioni, and M. Behnam, "Architecting Digital Twins," *IEEE Access*, vol. 10, pp. 50 335–50 350, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9770073

[60] J. Pfeiffer, J. Zhang, B. Combemale, J. Michael, B. Rumpe, M. Wimmer, and A. Wortmann, "Towards a Unifying Reference Model for Digital Twins of Cyber-Physical Systems," Jul. 2025, arXiv:2507.04871 [cs]. [Online]. Available: http://arxiv.org/abs/2507.04871

[61] ——, *Towards a Unifying Reference Model for Digital Twins of Cyber-Physical Systems*, Jul. 2025.

[62] N. Anwer, R. Stark, F. Tao, and J. Erkoyuncu, "Developing and leveraging digital twins in engineering design," *CIRP Annals*, vol. 2025, Jun. 2025.

[63] N. Anwer, R. Stark, F. Tao, and J. A. Erkoyuncu, "Developing and leveraging digital twins in engineering design," *CIRP Annals*, vol. 74, no. 2, pp. 843–868, Jan. 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0007850625001477

[64] M. Heithoff, N. Jansen, J. Michael, F. Rademacher, and B. Rumpe, "Model-Based Engineering of Multi-Purpose Digital Twins in Manufacturing," in *Digital Twin: Fundamentals and Applications*, S. Sabri, K. Alexandridis, and N. Lee, Eds. Cham: Springer Nature Switzerland, 2024, pp. 89–126. [Online]. Available: https://doi.org/10.1007/978-3-031-67778-6_5

[65] M. Heithoff, M. Trinh, J. Michael, B. Rumpe, and C. Brecher, "A Digital Shadow for Accurate Robot Motion Control: Integrating Data with Friction Models," Grand Rapids, USA, Jul. 2025. [Online]. Available: https://epub.uni-regensburg.de/77667/

[66] M. G. Kapteyn, D. J. Knezevic, and K. Willcox, "Toward predictive digital twins via component-based reduced-order models and interpretable machine learning," in *AIAA Scitech 2020 Forum*, ser. AIAA SciTech Forum. American Institute of Aeronautics and Astronautics, Jan. 2020. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2020-0418

[67] J. Liu, J. Vatn, and S. Yin, "A Case Study for Enhancing Maintenance of Cyber-Physical Systems With Digital Twin," *IEEE Transactions on Industrial Cyber-Physical Systems*, vol. 2, pp. 597–605, 2024, conference Name: IEEE Transactions on Industrial Cyber-Physical Systems. [Online]. Available: https://ieeexplore.ieee.org/document/10731633

[68] E. Kamburjan, N. Bencomo, S. L. Tapia Tarifa, and E. B. Johnsen, "Declarative Lifecycle Management in Digital Twins," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*. Linz Austria: ACM, Sep. 2024, pp. 353–363. [Online]. Available: https://dl.acm.org/doi/10.1145/3652620.3688248

[69] I. Aldalur, A. Arrieta, A. Agirre, G. Sagardui, and M. Arratibel, "A microservice-based framework for multi-level testing of cyber-physical systems," *Software Quality Journal*, vol. 32, no. 1, pp. 193–223, Mar. 2024. [Online]. Available: https://doi.org/10.1007/s11219-023-09639-z

[70] G. Lyan, J.-M. Jézéquel, D. Gross-Amblard, and B. Combemale, "DataTime: a Framework to smoothly Integrate Past, Present and Future into Models," in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Oct. 2021, pp. 134–144. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9592190

[71] C. Ellwein, D. Dietrich, N. Maisch, R. Neumann, S. Ajdinović, A. Lechler, and A. Wortmann, "Rethinking Asset Administration Shell Communication Types: A Systematic Mapping Study and Portfolio-Based Classification," *Production Engineering*, vol. 20, Dec. 2025.

[72] A. J. G. de Azambuja, T. Giese, K. Schützer, R. Anderl, B. Schleich, and V. R. Almeida, "Digital Twins in Industry 4.0 – Opportunities and challenges related to Cyber Security," *Procedia CIRP*, vol. 121, pp. 25–30, Jan. 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827123009563

[73] E. Katsoulakis, Q. Wang, H. Wu, L. Shahriyari, R. Fletcher, J. Liu, L. Achenie, H. Liu, P. Jackson, Y. Xiao, T. Syeda-Mahmood, R. Tuli, and J. Deng, "Digital twins for health: a scoping review," *npj Digital Medicine*, vol. 7, no. 1, p. 77, Mar. 2024. [Online]. Available: https://www.nature.com/articles/s41746-024-01073-0

[74] A. Osipov, O. Nikolic, A. Gertych, S. Parker, A. Hendifar, P. Singh, D. Filippova, G. Dagliyan, C. R. Ferrone, L. Zheng, J. H. Moore, W. Tourtellotte, J. E. Van Eyk, and D. Theodorescu, "The Molecular Twin artificial-intelligence platform integrates multi-omic data to predict outcomes for pancreatic adenocarcinoma patients," *Nature Cancer*, vol. 5, no. 2, pp. 299–314, Feb. 2024. [Online]. Available: https://www.nature.com/articles/s43018-023-00697-7

[75] S. Sabri, K. Alexandridis, and N. Lee, Eds., *Digital Twin: Fundamentals and Applications*. Cham: Springer Nature Switzerland, 2024. [Online]. Available: https://link.springer.com/10.1007/978-3-031-67778-6

[76] Q. Xu, T. Yue, S. Ali, and M. Arratibel, "Pretrain, Prompt, and Transfer: Evolving Digital Twins for Time-to-Event Analysis in Cyber-physical Systems," Apr. 2024, arXiv:2310.00032 [cs]. [Online]. Available: http://arxiv.org/abs/2310.00032

[77] C. Gomes, D. E. L. Rötter, A. Iosifidis, H. Feng, H. Ejersbo, and M. Frasheri, "Sensing and Communication of Data from the Physical Twin," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds. Cham: Springer International Publishing, 2024, pp. 147–171. [Online]. Available: https://doi.org/10.1007/978-3-031-66719-0_7

[78] ISO, "ISO/IEC 30141: Internet of things (IoT) – Reference architecture."

[79] C. Gomes, B. J. Oakes, J. Fitzgerald, and P. G. Larsen, "Foundational Concepts for Digital Twins of Cyber-Physical Systems," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds. Cham: Springer International Publishing, 2024, pp. 45–63. [Online]. Available: https://doi.org/10.1007/978-3-031-66719-0_3

[80] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *2017 IEEE international systems engineering symposium (ISSE)*. IEEE, 2017, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8088251/

[81] M. Picone, R. Morandi, A. Barbone, S. Burattini, M. Fogli, N. Bicocchi, C. Giannelli, and A. Ricci, "Harmonizing Physical and Digital Twins Lifecycles," in *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*, Mar. 2025, pp. 197–204. [Online]. Available: https://ieeexplore.ieee.org/document/11015033

[82] M. Picone, M. Mamei, and F. Zambonelli, "WLDT: A general purpose library to build IoT digital twins," *SoftwareX*, vol. 13, p. 100661, Jan. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711021000066

[83] ——, "A Flexible and Modular Architecture for Edge Digital Twin: Implementation and Evaluation," *ACM Transactions on Internet of Things*, vol. 4, no. 1, pp. 8:1–8:32, Feb. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3573206

[84] M. Martinelli, J. Zhang, A.-K. Splettstoßer, M. Picone, M. Lippi, and A. Wortmann, "Hierarchical Digital Twin Ecosystem for Industrial Manufacturing Scenarios," in *2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2024, pp. 56–63. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10803355

[85] N. Li, G. Xie, X. Zhang, H. Shi, X. Nie, Y. Wang, J. Wang, and J. Wang, "Six-dimensional digital twin modeling and software platform design for complex industrial systems," *Journal of Intelligent Manufacturing*, Jan. 2025. [Online]. Available: https://doi.org/10.1007/s10845-025-02567-8

[86] M. Kovatsch, M. Lanter, and Z. Shelby, "Californium: Scalable cloud services for the internet of things with coap," in *2014 International Conference on the Internet of Things (IOT)*. IEEE, 2014, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7030106/

[87] G. Shao, S. Frechette, and V. Srinivasan, "An Analysis of the New ISO 23247 Series of Standards on Digital Twin Framework for Manufacturing." American Society of Mechanical Engineers Digital Collection, Sep. 2023. [Online]. Available: https://dx.doi.org/10.1115/MSEC2023-101127

[88] T. L. Leirmo, "Digital Twins for Industry 5.0: Unlocking the Human Potential," *Procedia CIRP*, vol. 130, pp. 761–766, Jan. 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827124013180

[89] L. Esterle, M. Frasheri, and P. G. Larsen, "Autonomous Reconfiguration Enabled by Digital Twins," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds. Cham: Springer International Publishing, 2024, pp. 345–362. [Online]. Available: https://doi.org/10.1007/978-3-031-66719-0_14

[90] *Foundational Research Gaps and Future Directions for Digital Twins*. Washington, D.C.: National Academies Press, Mar. 2024. [Online]. Available: https://www.nap.edu/catalog/26894

[91] T. Böttjer, D. Tola, F. Kakavandi, C. R. Wewer, D. Ramanujan, C. Gomes, P. G. Larsen, and A. Iosifidis, "A review of unit level digital twin applications in the

manufacturing industry," *CIRP Journal of Manufacturing Science and Technology*, vol. 45, pp. 162–189, Oct. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1755581723000822

[92] E. VanDerHorn and S. Mahadevan, "Digital Twin: Generalization, characterization and implementation," *Decision Support Systems*, vol. 145, p. 113524, Jun. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167923621000348

[93] R. Kimmel, J. Michael, A. Wortmann, and J. Zhang, "Digital Twins for Software Engineering Processes," in *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, Apr. 2025, pp. 16–20. [Online]. Available: https://ieeexplore.ieee.org/document/11023962

[94] B. Combemale, J.-M. Jézéquel, Q. Perez, D. Vojtisek, N. Jansen, J. Michael, F. Rademacher, B. Rumpe, A. Wortmann, and J. Zhang, "Model-Based DevOps: Foundations and Challenges," in *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, Oct. 2023, pp. 429–433. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10350755

[95] J. Hugues, A. Hristosov, J. J. Hudak, and J. Yankel, "TwinOps - DevOps meets model-based engineering and digital twins for the engineering of CPS," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, ser. MODELS '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 1–5. [Online]. Available: https://dl.acm.org/doi/10.1145/3417990.3421446

[96] S. Aissat, J. Beaulieu, É. Poirier, A. Motamedi, J. Gascon-Samson, and F. Bordeleau, "A devops framework for the systematic engineering and evolution of digital twins for built assets," *Software and Systems Modeling*, Oct. 2025. [Online]. Available: https://doi.org/10.1007/s10270-025-01330-0

[97] C. Lin, T.-Y. Dai, A. D. Dilsiz, D. Crawley, D. Niyogi, and Z. Nagy, "UTwin: A digital twin of the UT Austin campus," in *Proceedings of the 10th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*. Istanbul Turkey: ACM, Nov. 2023, pp. 282–283. [Online]. Available: https://dl.acm.org/doi/10.1145/3600100.3626261

[98] R. Sieve, P. Kobialka, L. Slaughter, R. Schlatte, E. B. Johnsen, and S. L. T. Tarifa, "BedreFlyt: Improving Patient Flows through Hospital Wards with Digital Twins," *Electronic Proceedings in Theoretical Computer Science*, vol. 418, pp. 1–15, May 2025, arXiv:2505.06287 [cs]. [Online]. Available: http://arxiv.org/abs/2505.06287

[99] S. Burattini, S. Montagna, A. Croatti, N. Gentili, A. Ricci, L. Leonardi, S. Pandolfini, and S. Tosi, "An Ecosystem of Digital Twins for Operating Room Management," in *2023 IEEE 36th International Symposium on Computer-Based Medical Systems (CBMS)*, Jun. 2023, pp. 770–775. [Online]. Available: https://ieeexplore.ieee.org/document/10178878

[100] J. Li, J. Grübel, A. Nadi, M. Snelder, B. van Arem, and J. Gao, "Digital Twin Federation for Urban Mobility Assessment: A Functional Architecture for Low-Car Transformations in the Netherlands," 2025. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5370630

[101] M. Liu, S. Fang, H. Dong, and C. Xu, "Review of digital twin about concepts, technologies, and industrial applications," *Journal of Manufacturing Systems*, vol. 58, pp.

346–361, Jan. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0278612520301072

[102] "Digital Twins along the product lifecycle: A systematic literature ..." Mar. 2023. [Online]. Available: https://digitaltwin1.org/articles/3-3

[103] J. Michael, L. Cleophas, S. Zschaler, T. Clark, B. Combemale, T. Godfrey, D. E. Khelladi, V. Kulkarni, D. Lehner, B. Rumpe, M. Wimmer, A. Wortmann, S. Ali, B. Barn, I. Barosan, N. Bencomo, F. Bordeleau, G. Grossmann, G. Karsai, O. Kopp, B. Mitschang, P. Muñoz Ariza, A. Pierantonio, F. A. C. Polack, M. Riebisch, H. Schlingloff, M. Stumptner, A. Vallecillo, M. van den Brand, and H. Vangheluwe, "Model-Driven Engineering for Digital Twins: Opportunities and Challenges," *Systems Engineering*, vol. 28, no. 5, pp. 659–670, 2025, _eprint: https://incose.onlinelibrary.wiley.com/doi/pdf/10.1002/sys.21815. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sys.21815

[104] R. Laubenbacher, A. Niarakis, T. Helikar, G. An, B. Shapiro, R. S. Malik-Sheriff, T. J. Sego, A. Knapp, P. Macklin, and J. A. Glazier, "Building digital twins of the human immune system: toward a roadmap," *npj Digital Medicine*, vol. 5, no. 1, pp. 1–5, May 2022. [Online]. Available: https://www.nature.com/articles/s41746-022-00610-z

[105] F. Emmert-Streib, S. Parkkila, R. Laubenbacher, A. Mannermaa, L. Hood, and O. Yli-Harja, "The role of digital twins in P4 medicine: A paradigm for modern healthcare," *npj Digital Medicine*, vol. 8, no. 1, p. 735, Dec. 2025. [Online]. Available: https://www.nature.com/articles/s41746-025-02115-x

[106] Y. Ye, A. González-Vidal, A. Cisterna-García, A. Pérez-Ruzafa, M. A. Z. Izquierdo, and A. F. Skarmeta, "Advancing Towards a Marine Digital Twin Platform: Modeling the Mar Menor Coastal Lagoon Ecosystem in the South Western Mediterranean," Sep. 2024, arXiv:2409.10134 [cs]. [Online]. Available: http://arxiv.org/abs/2409.10134

[107] "interTwin: Advancing Scientific Digital Twins through AI, Federated Computing and Data."

[108] "The rise of digital twins," Mar. 2024. [Online]. Available: https://www.nature.com/collections/feicjiideh

[109] Z. Wang, R. Gupta, K. Han, H. Wang, A. Ganlath, N. Ammar, and P. Tiwari, "Mobility digital twin: Concept, architecture, case study, and future challenges," *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 17 452–17 467, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9724183/

[110] C. H. Bohlbro, H. D. Macedo, D. Tola, L. Esterle, and P. G. Larsen, "Visualisation in a Digital Twin Context," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds. Cham: Springer International Publishing, 2024, pp. 175–188. [Online]. Available: https://doi.org/10.1007/978-3-031-66719-0_8

[111] X. Liu and I. David, "AI simulation by digital twins: systematic survey, reference framework, and mapping to a standardized architecture," *Software and Systems Modeling*, Aug. 2025. [Online]. Available: https://doi.org/10.1007/s10270-025-01306-0

[112] P. Bellavista, N. Bicocchi, M. Fogli, C. Giannelli, M. Mamei, and M. Picone, "Exploiting microservices and serverless for Digital Twins in the cloud-to-edge continuum," *Future Generation Computer Systems*, vol. 157, pp. 275–287, Aug. 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X24001249

[113] M. Picone, E. Bouquillon, M. Pietri, and M. Mamei, "Assessing the Impact of Cybersecurity Attacks on Digital Twin Metrics: An Experimental Study," in *2026 IEEE 23rd Consumer Communications & Networking Conference (CCNC)*. IEEE, 2026, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11366535/

[114] P. Talasila, P. H. Mikkelsen, S. Gil, and P. G. Larsen, "Realising Digital Twins," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds. Cham: Springer International Publishing, 2024, pp. 225–256. [Online]. Available: https://doi.org/10.1007/978-3-031-66719-0_11

[115] A. Ricci, A. Croatti, S. Mariani, S. Montagna, and M. Picone, "Web of Digital Twins," *ACM Transactions on Internet Technology*, vol. 22, no. 4, pp. 1–30, Nov. 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3507909

[116] S. Gil, B. Oakes, C. Gomes, M. Frasheri, and P. Larsen, "Toward a systematic reporting framework for Digital Twins: a cooperative robotics case study," *SIMULATION*, Aug. 2024.

[117] G. Schweiger, C. Gomes, G. Engel, I. Hafner, J. Schoeggl, A. Posch, and T. Nouidui, "An empirical survey on co-simulation: Promising standards, challenges and research needs," *Simulation Modelling Practice and Theory*, vol. 95, pp. 148–163, Sep. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1569190X1930053X

[118] Z. Liu, Y. Chu, G. Li, H. P. Hildre, and H. Zhang, "Shipboard crane digital twin: An empirical study on R/V Gunnerus," *Ocean Engineering*, vol. 302, p. 117675, Jun. 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0029801824010126

[119] S. Gil, P. H. Mikkelsen, C. Gomes, and P. G. Larsen, "Survey on open-source digital twin frameworks–A case study approach," *Software: Practice and Experience*, vol. 54, no. 6, pp. 929–960, Jun. 2024. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/spe.3305

[120] C. Friedrich, A. Lombana, J. Fasquel, C. Schlick, N. Bennani, and M. Mendil, "CoFMPy: A Python Framework for Rapid Prototyping of FMI-based Digital Twins," in *The 2nd International Conference on Engineering Digital Twins*, 2025. [Online]. Available: https://hal.science/hal-05326255/

[121] K. Duran, L. Verda Cakir, Y. Yigit, K. Huseynov, S. Ram Kusu, M. Ali Ertürk, and B. Canberk, "Toward Digital Twin-as-a-Service (DTaaS) Platforms: A Survey on Architecture, Design Requirements, and Performance Metrics," *IEEE Communications Surveys & Tutorials*, vol. 28, pp. 1845–1878, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/11261676

[122] P. Bellavista, N. Bicocchi, M. Fogli, C. Giannelli, M. Mamei, and M. Picone, "An Entanglement-Aware Middleware for Digital Twins," *ACM Trans. Internet Things*, Oct. 2024, just Accepted. [Online]. Available: https://dl.acm.org/doi/10.1145/3699520

[123] F. Tao, W. Liu, M. Zhang, T.-l. Hu, Q. Qi, H. Zhang, F. Sui, T. Wang, H. Xu, and Z. Huang, "Five-dimension digital twin model and its ten applications," *Comput. Integr. Manuf. Syst*, vol. 25, no. 1, pp. 1–18, 2019.

[124] F. Tao, X. Sun, J. Cheng, Y. Zhu, W. Liu, Y. Wang, H. Xu, T. Hu, X. Liu, T. Liu, Z. Sun, J. Xu, J. Bao, F. Xiang, and X. Jin, "makeTwin: A reference architecture for digital twin software platform," *Chinese Journal of Aeronautics*, vol. 37, no. 1, pp. 1–18, Jan. 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1000936123001541

[125] G. Shao, "Use Case Scenarios for Digital Twin Implementation Based on ISO 23247," *NIST*, May 2021, last Modified: 2022-11-29T09:11-05:00. [Online]. Available: https://www.nist.gov/publications/use-case-scenarios-digital-twin-implementation-based-iso-23247

[126] L. Heaton, "Platform Stack Architectural Framework: An Introductory Guide."

[127] kayla, "Digital Twin Platform Stack Architectural Framework by Digital Twin Consortium," Nov. 2023. [Online]. Available: https://www.digitaltwinconsortium.org/2023/11/understanding-dtcs-digital-twin-platform-stack-architectural-framework/

[128] "Glossary of Digital Twins by Digital Twin Consortium." [Online]. Available: https://www.digitaltwinconsortium.org/glossary/glossary/

[129] A. Budiardjo and D. Migliori, "Digital twin system interoperability framework," Tech. rep. Digital Twin Consortium, East Lansing, Michigan, Tech. Rep., 2021. [Online]. Available: https://www.digitaltwinconsortium.org/pdf/Digital-Twin-System-Interoperability-Framework-12072021.pdf

[130] A. Giulianelli, S. Burattini, A. Ciortea, and A. Ricci, "HWoDT Framework: A toolchain to build interoperable Digital Twin Ecosystems," *SoftwareX*, vol. 31, p. 102275, Sep. 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711025002420

[131] J. Zhang, C. Ellwein, M. Heithoff, J. Michael, and A. Wortmann, "Digital twin and the asset administration shell," 2024. [Online]. Available: https://awortmann.github.io/downloads/paper/Digital_twin_and_the_asset_administration_shell.pdf

[132] J. Robles, C. Martín, and M. Díaz, "OpenTwins: An open-source framework for the development of next-gen compositional digital twins," *Computers in Industry*, vol. 152, p. 104007, Aug. 2023.

[133] J. Mattila, R. Ala-Laurinaho, J. Autiosalo, and K. Tammi, "Interoperability of Digital Twins for Automation With Digital Twin Schema," *IEEE Access*, vol. 13, pp. 200 595–200 608, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/11251231

[134] S. Burattini, A. Zimmermann, M. Picone, and A. Ricci, "Towards Linked Data for Ecosystems of Digital Twins," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS Companion '24. New York, NY, USA: Association for Computing Machinery, Oct. 2024, pp. 332–337. [Online]. Available: https://dl.acm.org/doi/10.1145/3652620.3688245

[135] A. Giulianelli, S. Burattini, A. Ciortea, and A. Ricci, "Engineering Interoperable Ecosystems of Digital Twins: A Web-based Approach," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS Companion '24. New York, NY, USA: Association for Computing Machinery, Oct. 2024, pp. 476–485. [Online]. Available: https://dl.acm.org/doi/10.1145/3652620.3688263

[136] G. Beaumont, A. Beugnard, S. Martínez, C. Urtado, and S. Vauttier, "Towards Automating the Life Cycle Management of Digital Twins," in *ER2025-44th International Conference on Conceptual Modeling*, 2025. [Online]. Available: https://hal.science/hal-05225783/

[137] A. Almeida, T. Batista, E. Cavalcante, F. Delicato, R. Motta, and M. Vieira, "Middleware for Digital Twins: A Systematic Mapping Study," in *Proceedings of the 1st International Workshop on Middleware for Digital Twin*, ser. Midd4DT '23. New York, NY, USA:

Association for Computing Machinery, Dec. 2023, pp. 19–24. [Online]. Available: https://dl.acm.org/doi/10.1145/3631319.3632302

[138] A. C. Marosi, M. Emodi, Á. Hajnal, R. Lovas, T. Kiss, V. Poser, J. Antony, S. Bergweiler, H. Hamzeh, J. Deslauriers, and J. Kovács, "Interoperable Data Analytics Reference Architectures Empowering Digital-Twin-Aided Manufacturing," *Future Internet*, vol. 14, no. 4, p. 114, Apr. 2022, number: 4. [Online]. Available: https://www.mdpi.com/1999-5903/14/4/114

[139] P.-E. Goffi, R. Tremblay, and B. Oakes, "Engineering a Digital Twin for the Monitoring and Control of Beer Fermentation Sampling," Aug. 2025, arXiv:2508.18452 [cs]. [Online]. Available: http://arxiv.org/abs/2508.18452

[140] C. Savaglio, V. Barbuto, F. Mangione, and G. Fortino, "Generative Digital Twins: A Novel Approach in the IoT Edge-Cloud Continuum," *IEEE Internet of Things Magazine*, vol. 8, no. 1, pp. 42–48, Jan. 2025, conference Name: IEEE Internet of Things Magazine. [Online]. Available: https://ieeexplore.ieee.org/document/10638534

[141] M. Frasheri, H. Ejersbo, C. Thule, C. Gomes, J. L. Kvistgaard, P. G. Larsen, and L. Esterle, "Addressing time discrepancy between digital and physical twins," *Robotics and Autonomous Systems*, vol. 161, p. 104347, Mar. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889022002366

[142] B. Mishra and A. Kertesz, "The use of MQTT in M2M and IoT systems: A survey," *Ieee Access*, vol. 8, pp. 201 071–201 086, 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9247996/

[143] A. R. Qureshi, A. Asensio, M. Imran, J. Garcia, and X. Masip-Bruin, "A survey on security enhancing Digital Twins: Models, applications and tools," *Computer Communications*, vol. 238, p. 108158, Jun. 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S014036642500115X

[144] S. Cavalieri and S. Gambadoro, "Proposal of Mapping Digital Twins Definition Language to Open Platform Communications Unified Architecture," *Sensors*, vol. 23, p. 2349, Feb. 2023.

[145] Q.-D. Nguyen, Y. Huang, F. Keith, C. Leroy, M.-T. Thi, and S. Dhouib, "Manufacturing 4.0: Checking the Feasibility of a Work Cell Using Asset Administration Shell and Physics-Based Three-Dimensional Digital Twins," *Machines*, vol. 12, p. 95, Jan. 2024.

[146] A. Lazidis, K. Tsakos, and E. Petrakis, "Publish-Subscribe approaches for the IoT and the cloud: Functional and performance evaluation of open-source systems," *Internet of Things*, vol. 19, p. 100538, May 2022.

[147] "Summary of IoT, and DT Standards."

[148] "AutomationML: A Practical Guide," in *AutomationML*. De Gruyter Oldenbourg, Jul. 2021. [Online]. Available: https://www.degruyter.com/document/doi/10.1515/9783110746235/html

[149] R. Minerva, G. M. Lee, and N. Crespi, "Digital twin in the IoT context: A survey on technical features, scenarios, and architectural models," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1785–1824, 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9120192/

[150] L. Bedogni, M. Mamei, M. Picone, M. Pietri, and F. Zambonelli, "Fluid Computing & Digital Twins for intelligent interoperability in the IoT ecosystem," *Future Generation Computer Systems*, vol. 171, p. 107855, Oct. 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X25001505

[151] S. Deng, H. Zhao, B. Huang, C. Zhang, F. Chen, Y. Deng, J. Yin, S. Dustdar, and A. Y. Zomaya, "Cloud-Native Computing: A Survey From the Perspective of Services," *Proceedings of the IEEE*, vol. 112, no. 1, pp. 12–46, Jan. 2024, conference Name: Proceedings of the IEEE. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10433234

[152] S. M. Raza, R. Minerva, B. Martini, and N. Crespi, "Empowering Microservices: A Deep Dive into Intelligent Application Component Placement for Optimal Response Time," *Journal of Network and Systems Management*, vol. 32, no. 4, p. 84, Aug. 2024. [Online]. Available: https://doi.org/10.1007/s10922-024-09855-3

[153] R. Singh, J. Kovacs, and T. Kiss, "To Offload or Not? An Analysis of Big Data Offloading Strategies from Edge to Cloud," in *2022 IEEE World AI IoT Congress (AIIoT)*, Jun. 2022, pp. 046–052. [Online]. Available: https://ieeexplore.ieee.org/document/9817276

[154] S. K. Jensen, C. S. Godiksen, C. Thomsen, and T. B. Pedersen, "Demonstration of ModelarDB: Model-Based Management of High-Frequency Time Series Across Edge, Cloud, and Client," *Proceedings of the VLDB Endowment*, vol. 18, no. 12, pp. 5247–5250, Aug. 2025. [Online]. Available: https://dl.acm.org/doi/10.14778/3750601.3750643

[155] S. K. Jensen, C. Thomsen, and T. B. Pedersen, "ModelarDB: Integrated Model-Based Management of Time Series from Edge to Cloud," in *Transactions on Large-Scale Data- and Knowledge-Centered Systems LIII*, A. Hameurlain and A. M. Tjoa, Eds. Berlin, Heidelberg: Springer, 2023, pp. 1–33. [Online]. Available: https://doi.org/10.1007/978-3-662-66863-4_1

[156] A. Abduvakhobov, S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Scalable Model-Based Management of Massive High Frequency Wind Turbine Data with ModelarDB," *Proceedings of the VLDB Endowment*, vol. 17, no. 13, pp. 4723–4732, Sep. 2024. [Online]. Available: https://dl.acm.org/doi/10.14778/3704965.3704978

[157] J. Vaillant, I. Grechi, F. Normand, and F. Boudon, "Towards virtual modelling environments for functional–structural plant models based on Jupyter notebooks: application to the modelling of mango tree growth and development," *in silico Plants*, vol. 4, no. 1, p. diab040, Jan. 2022. [Online]. Available: https://doi.org/10.1093/insilicoplants/diab040

[158] J. Slavič, K. Zaletelj, D. Gorjup, D. P. Pasca, A. Aloisio, K. A. Kvåle, G. T. Frøseth, W. Weijtjens, A. Mujtaba, M. Česnik, A. Zorman, G. Tsialiamanis, I. Tomac, T. G. Ritto, D. Ocepek, D. Rohe, O. M. Silva, T. C. P. Masmeijer, K. Čufar, T. Bregar, M. Kodrič, A. El Mahmoudi, M. Pogačar, F. Trainotti, M. Göldeli, G. Čepon, R. Timbó, A. A. Cavallini, V. T. Costa, J. G. F. Santos, A. J. Hughes, J. G. Vargas, A. Fernandes, D. J. Rixen, K. Worden, and M. Boltežar, "Python open-source signal processing and modeling/simulation for scientific research in structural dynamics—A review," *Mechanical Systems and Signal Processing*, vol. 241, p. 113465, Dec. 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0888327025011665

[159] J. Chae, S. Lee, J. Jang, S. Hong, and K.-J. Park, "A Survey and Perspective on Industrial Cyber-Physical Systems (ICPS): From ICPS to AI-Augmented ICPS," *IEEE Transactions*

on Industrial Cyber-Physical Systems, vol. 1, pp. 257–272, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10285426

[160] C. Badolato, N. Samson, H. H. Hassan, C.-K. Huang, G. Bouloukakis, P. Pappachan, and R. Yus, "PSMark: a distributed IoT benchmark for publish/subscribe under domain-based workloads," in *PerCom 2026-24th IEEE International Conference on Pervasive Computing and Communications*, 2026. [Online]. Available: https://inria.hal.science/hal-05517145/

[161] M. Iorio, F. Risso, and C. Casetti, "When latency matters: measurements and lessons learned," *SIGCOMM Comput. Commun. Rev.*, vol. 51, no. 4, pp. 2–13, Dec. 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3503954.3503956

[162] L. A. Bull, I. Abdallah, C. Mylonas, L. D. Avendaño-Valencia, K. Tatsis, P. Gardner, T. J. Rogers, D. S. Brennan, E. J. Cross, and K. Worden, "Data-Centric Monitoring of Wind Farms: Combining Sources of Information," in *Data Driven Methods for Civil Structural Health Monitoring and Resilience*. CRC Press, 2023, num Pages: 61.

[163] A. Thelen, X. Huan, N. Paulson, S. Onori, Z. Hu, and C. Hu, "Probabilistic machine learning for battery health diagnostics and prognostics—review and perspectives," *npj Materials Sustainability*, vol. 2, no. 1, pp. 1–33, Jun. 2024. [Online]. Available: https://www.nature.com/articles/s44296-024-00011-1

[164] E. Branlard, J. Jonkman, C. Brown, and J. Zhang, "A digital twin solution for floating offshore wind turbines validated using a full-scale prototype," *Wind Energy Science*, vol. 9, no. 1, pp. 1–24, Jan. 2024. [Online]. Available: https://wes.copernicus.org/articles/9/1/2024/

[165] U. T. Tygesen, K. Worden, T. Rogers, G. Manson, and E. J. Cross, "State-of-the-Art and Future Directions for Predictive Modelling of Offshore Structure Dynamics Using Machine Learning," in *Dynamics of Civil Structures, Volume 2*, S. Pakzad, Ed. Cham: Springer International Publishing, 2019, pp. 223–233.

[166] P. Carreira, V. Amaral, and H. Vangheluwe, Eds., *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*. Springer Nature, 2020, accepted: 2020-05-13T14:21:18Z. [Online]. Available: https://library.oapen.org/handle/20.500.12657/37711

[167] J. Hugues, J. Yankel, J. Hudak, and A. Hristozov, "Twinops: Digital twins meets devops," *CARNEGIE-MELLON UNIV PITTSBURGH PA, Tech. Rep.*, 2022. [Online]. Available: https://apps.dtic.mil/sti/trecms/pdf/AD1168471.pdf

[168] D. Shangguan, L. Chen, C. Su, J. Ding, and C. Liu, "A Triple Human-Digital Twin Architecture for Cyber-Physical Systems," *Computer Modeling in Engineering & Sciences*, vol. 131, no. 3, pp. 1557–1578, 2022. [Online]. Available: https://www.techscience.com/CMES/v131n3/47386

[169] M. Wetter, K. Benne, H. Tummescheit, and C. Winther, "Spawn: coupling Modelica Buildings Library and EnergyPlus to enable new energy system and control applications," *Journal of Building Performance Simulation*, vol. 17, no. 2, pp. 274–292, Mar. 2024, _eprint: https://doi.org/10.1080/19401493.2023.2266414. [Online]. Available: https://doi.org/10.1080/19401493.2023.2266414

[170] M. Dalibor, N. Jansen, B. Rumpe, D. Schmalzing, L. Wachtmeister, M. Wimmer, and A. Wortmann, "A Cross-Domain Systematic Mapping Study on Software Engineering for Digital Twins," *Journal of Systems and Software*, vol. 193, p. 111361, Nov. 2022.

[171] M. G. Kapteyn, J. V. R. Pretorius, and K. E. Willcox, "A probabilistic graphical model foundation for enabling predictive digital twins at scale," *Nature Computational Science*, vol. 1, no. 5, pp. 337–347, May 2021. [Online]. Available: https://www.nature.com/articles/s43588-021-00069-0

[172] N. Wagner and D. Tcherniak, *Data-Driven Updating of Digital Twins through Experimental Measurements and Parametric Finite Element Model Optimization*, Nov. 2025.

[173] V. Zambrano, I. Viejo, G. López, J. Alfonso, J. Rodríguez, M. López-Blanco, P. Talasila, s. Calvo, and M. Sánchez, *Interpretable and Reusable Reduced Order Models for Digital Twins in Manufactory as a Service*, Jan. 2023.

[174] M. Torzoni, A. Manzoni, and S. Mariani, "A Deep Neural Network, Multi-fidelity Surrogate Model Approach for Bayesian Model Updating in SHM," in *European Workshop on Structural Health Monitoring*, P. Rizzo and A. Milazzo, Eds. Cham: Springer International Publishing, 2023, pp. 1076–1086.

[175] M. Torzoni, M. Tezzele, S. Mariani, A. Manzoni, and K. E. Willcox, "A digital twin framework for civil engineering structures," *Computer Methods in Applied Mechanics and Engineering*, vol. 418, p. 116584, Jan. 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045782523007089

[176] D. Lehner, A. Garmendia, and M. Wimmer, "Towards Flexible Evolution of Digital Twins with Fluent APIs," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2021, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9613222

[177] A. Thelen, M. Li, C. Hu, E. Bekyarova, S. Kalinin, and M. Sanghadasa, "Augmented model-based framework for battery remaining useful life prediction," *Applied Energy*, vol. 324, p. 119624, Oct. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306261922009266

[178] D. Lehner, S. Sint, M. Eisenberg, and M. Wimmer, "A pattern catalog for augmenting Digital Twin models with behavior," *at - Automatisierungstechnik*, vol. 71, no. 6, pp. 423–443, Jun. 2023. [Online]. Available: https://www.degruyter.com/document/doi/10.1515/auto-2022-0144/html

[179] Y. Ma, K. Younis, B. S. Ahmed, A. Kassler, P. Krakhmalev, A. Thore, and H. Lindback, "Automated and Systematic Digital Twins Testing for Industrial Processes," Feb. 2023, arXiv:2302.13198. [Online]. Available: http://arxiv.org/abs/2302.13198

[180] T. Li, J. Liu, A. Thelen, A. K. Mishra, X.-G. Yang, Z. Wang, and C. Hu, "Coupling a capacity fade model with machine learning for early prediction of the battery capacity trajectory," *Applied Energy*, vol. 389, p. 125703, Jul. 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306261925004337

[181] C. Cimino, F. Terraneo, G. Ferretti, and A. Leva, "Efficient Control Representation in Digital Twins: An Imperative Challenge for Declarative Languages," *IEEE Transactions on Industrial Informatics*, vol. 19, no. 11, pp. 11 080–11 090, Nov. 2023, conference Name: IEEE Transactions on Industrial Informatics. [Online]. Available: https://ieeexplore.ieee.org/document/10040242

[182] A. Wooley, J. Bitencourt, and D. Silva, "Bridging the gap between discrete event simulation and digital twin: A manufacturing case study☆☆," *Manufacturing Letters*,

vol. 44, pp. 1274–1284, Aug. 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2213846325001798

[183] A. Villalonga, E. Negri, G. Biscardo, F. Castano, R. E. Haber, L. Fumagalli, and M. Macchi, "A decision-making framework for dynamic scheduling of cyber-physical production systems based on digital twins," *Annual Reviews in Control*, vol. 51, pp. 357–373, Jan. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1367578821000262

[184] D. Moreau, K. Wiebels, and C. Boettiger, "Containers for computational reproducibility," *Nature Reviews Methods Primers*, vol. 3, no. 1, p. 50, 2023. [Online]. Available: https://www.nature.com/articles/s43586-023-00236-9

[185] A. Chaudhuri, G. Pash, D. A. I. Hormuth, G. Lorenzo, M. Kapteyn, C. Wu, E. A. B. F. Lima, T. E. Yankeelov, and K. Willcox, "Predictive digital twin for optimizing patient-specific radiotherapy regimens under uncertainty in high-grade gliomas," *Frontiers in Artificial Intelligence*, vol. 6, Oct. 2023. [Online]. Available: https://www.frontiersin.org/articles/10.3389/frai.2023.1222612

[186] A. Rasheed, O. San, and T. Kvamsdal, "Digital Twin: Values, Challenges and Enablers From a Modeling Perspective," *IEEE Access*, vol. 8, pp. 21 980–22 012, 2020, conference Name: IEEE Access. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8972429

[187] L. Esterle, C. Gomes, M. Frasheri, H. Ejersbo, S. Tomforde, and P. G. Larsen, "Digital twins for collaboration and self-integration," in *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 2021, pp. 172–177. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9599295/

[188] J. Grübel, C. V. Rios, C. Zuo, S. Ossey, R. M. Franken, M. Balac, Y. Xin, K. W. Axhausen, M. Raubal, and O. Riba-Grognuz, "Outlining the Open Digital Twin Platform," in *2023 IEEE Smart World Congress (SWC)*, Aug. 2023, pp. 1–3. [Online]. Available: https://ieeexplore.ieee.org/document/10448743

[189] R. Paredis, C. Gomes, and H. Vangheluwe, "A Family of Digital T Workflows and Architectures: Exploring Two Cases," in *Innovative Intelligent Industrial Production and Logistics*, A. Smirnov, H. Panetto, and K. Madani, Eds. Cham: Springer Nature Switzerland, 2023, pp. 93–109.

[190] H. Feng, C. Gomes, and P. G. Larsen, "Model-Based Monitoring and State Estimation for Digital Twins: The Kalman Filter," Apr. 2023, arXiv:2305.00252 [eess]. [Online]. Available: http://arxiv.org/abs/2305.00252

[191] F. Zonzini, C. Aguzzi, L. Gigli, L. Sciullo, N. Testoni, L. De Marchi, M. Di Felice, T. S. Cinotti, C. Mennuti, and A. Marzani, "Structural Health Monitoring and Prognostic of Industrial Plants and Civil Structures: A Sensor to Cloud Architecture," *IEEE Instrumentation & Measurement Magazine*, vol. 23, no. 9, pp. 21–27, Dec. 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9289069