Chapter 4: Digital Twin Architecture

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.

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 [1].

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 [2], [3] – 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 [4], and the model is where most of that evolution lands.

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 [5]. 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 [3], [6]. 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 [5]. Hold both facts loosely for now. The next section proposes a division of functions; the Architectural and Design Principles section returns to the patterns.

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 [7], and in which a twin is an assembly of services built upwards from models and data to analytics, visualisation and what-if analysis [8]. 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 [2], [9]. 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 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.

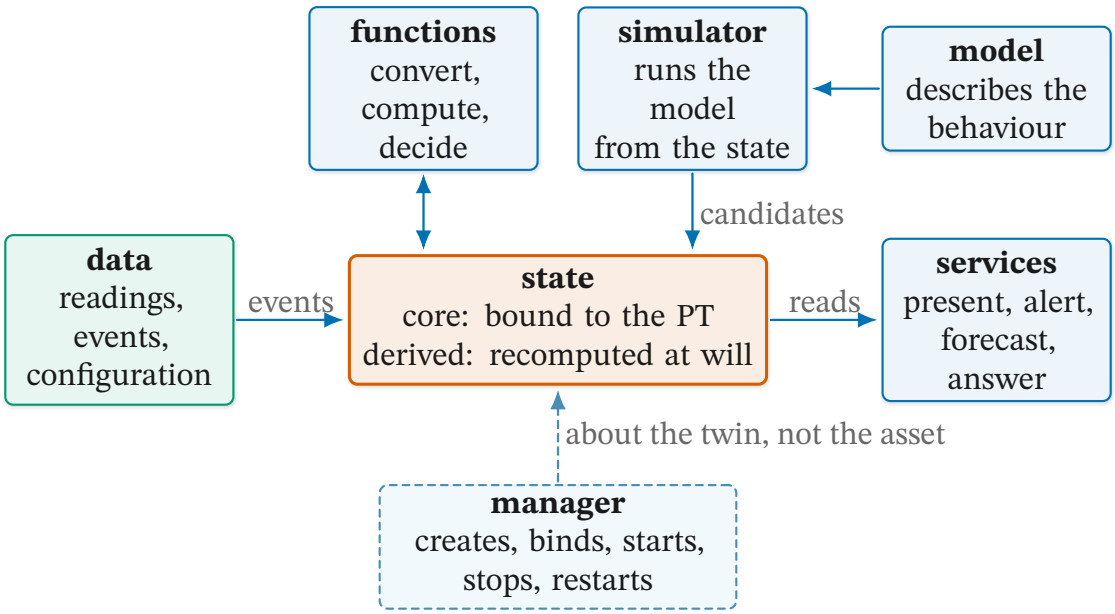


Figure 1: The seven building blocks, drawn around the one they all depend on. Data arrives and is recorded; functions and the model, run by a simulator, read and write the state; services read it; the manager stands outside the loop and is about the twin itself. Nothing reaches the state except through a block.

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 [10]. 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 [11]. 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.

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 [9]. 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 [12]; the two fields are how a consumer can tell whether the mirror is currently showing anything.

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 [7]. 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.

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 [13]; 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 [14]. Neither system rebuilt its twin to change the model, and that is the property the slot exists to give you.

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 [15]. A shipboard crane's twin, for instance, is built as a co-simulation through the FMI standard [16]. 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 [17], and a lighter alternative is a Python environment whose whole job is FMU execution [18]. 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.

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 [2]. 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 [9]. 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 [7]. 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.

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 [8]; 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.

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 [19] – 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 [8], 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 [20]. 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 1 gives each family a generic instance and the value patterns it delivers.

Table 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
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 [21] 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.

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 [2]. 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 [6]. 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 [22]. 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.

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 [3].

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 [23]. The rule has a corollary from the State subsection: the core part of that truth is written only by shadowing, never by an application [9].

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 [13], and a framework in which FMI services and learned models sit behind the same twin [14].

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 [3].

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 [7]. 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 [2] 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 [9] – 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 [24]. 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 [6]. A later chapter has more to say about security; this principle is the one architectural

decision it needs made now.

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 [5]. 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 [25] – 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 [26]. 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.

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.

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 [2]. 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 [27]; 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 [6]. Three independent statements of one structure, from one line of work, all of them compiled and measured. Figure 2 draws it.

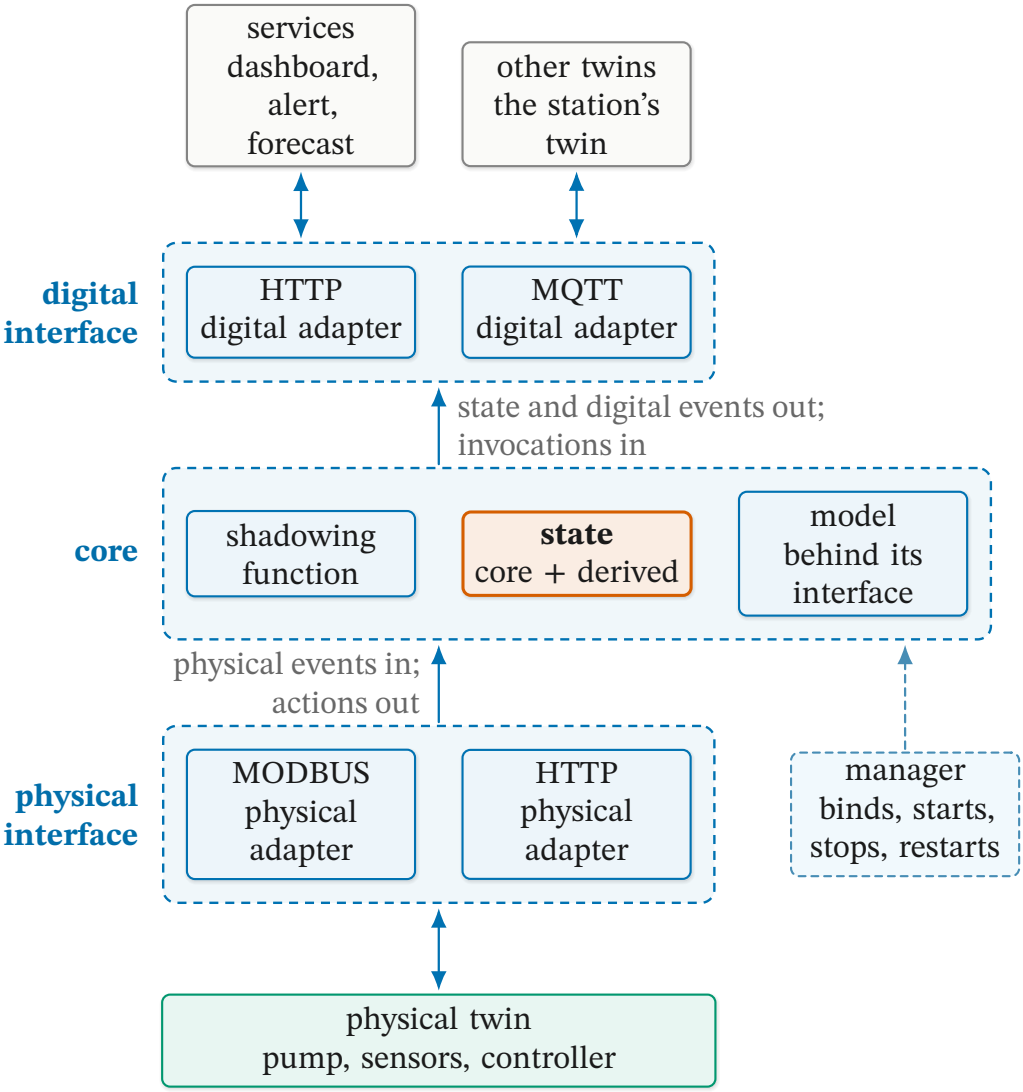


Figure 2: The three layers of a digital object. Physical events climb through the physical adapters into the core, where the shadowing function, the model and the state live; digital events and state leave through the digital adapters to services and other twins. Nothing crosses from the digital side to the hardware except through the core.

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 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 [3]. 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 [3], [9].

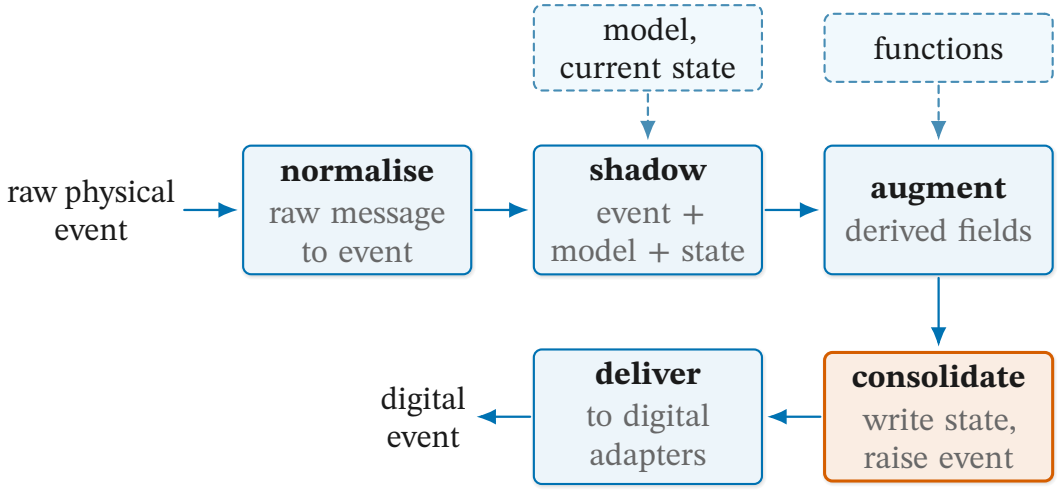


Figure 3: One reading through the shadowing pipeline. The raw message becomes a normalised event; the shadowing function turns the event, the model and the current state into a candidate; augmentation adds derived fields; consolidation writes the new state and raises a digital event; the digital adapters deliver it.

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.

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 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 [9]. 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 [1]. 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.

Every block on a layer

Table 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 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
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 [2]. 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.

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 [28], 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 [5]. One ships as a library. The lessons differ accordingly.

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 [29]. 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 [30].

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.

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 [19]. A companion paper walks use-case scenarios through the same functional-entity view [31].

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 [11] – which is the seven blocks' Data subsection arriving from the standards side.

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 [32]. 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 [33], 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 [34]. A separate consortium paper takes the systems-of-systems view, proposing an interoperability framework for plugging multi-vendor services into a system of twins [35].

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 [36], which is the same lesson in different words.

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 [2]. It is also the basis for the Web of Digital Twins ecosystem work, which uses it as one of the technologies its adapters wrap [37], and a survey of twin frameworks describes it as running at the edge or in the cloud and supporting a wide range of protocols [23]. The middleware work built on it observes what it lacks: a management layer for deploying and orchestrating many twins in a structured, observable way [24] – 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.

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 [38]. 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.

Reading the five together

Table 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 [28]; 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 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

This chapter	Five-dimension	ISO 23247	DTC platform stack	WLDT	Six-dimension
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 representation 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 [39], and so a candidate for the schema rung of the next section rather than a sixth column here.

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 is the ladder.

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.

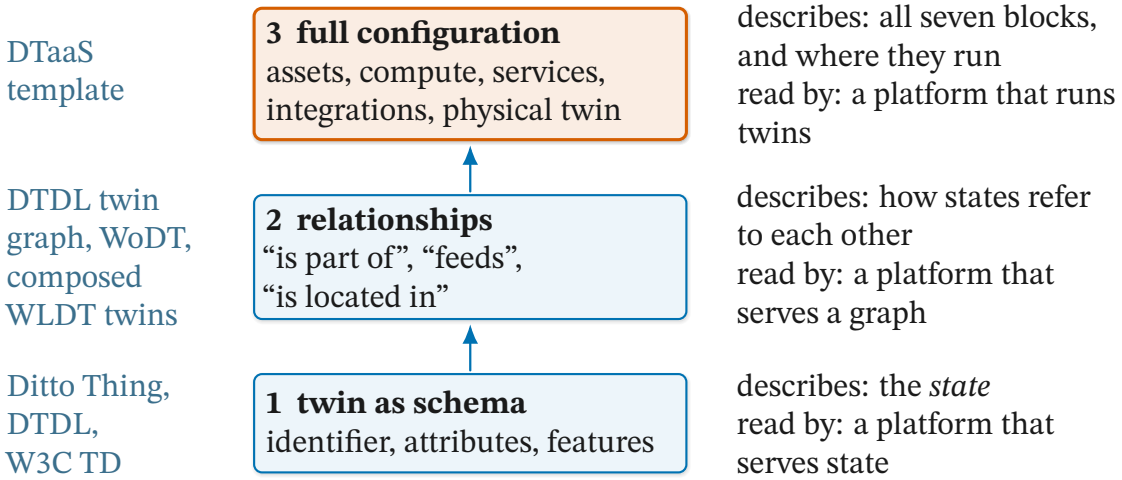


Figure 4: The three rungs of a twin’s configuration. Each rung describes more of the twin than the one below it, and each is what a different kind of platform reads: a schema is enough for a platform that serves state, relationships are enough for one that serves a graph, and a full configuration is what a platform that creates and runs twins needs.

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 [17] – 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 [40]. 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 [41]. 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.

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* [42]. 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 [9], [43]. 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 [12]. 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 [37].

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.

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 [7]. 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 [44].

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 [7]. 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 [45]. 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.

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 [46]. Nothing below is a toy argument about a toy.

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 places them.

Table 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

Block	On the rig	Where it came from
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.

The three layers on the rig

Figure 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.

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.

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

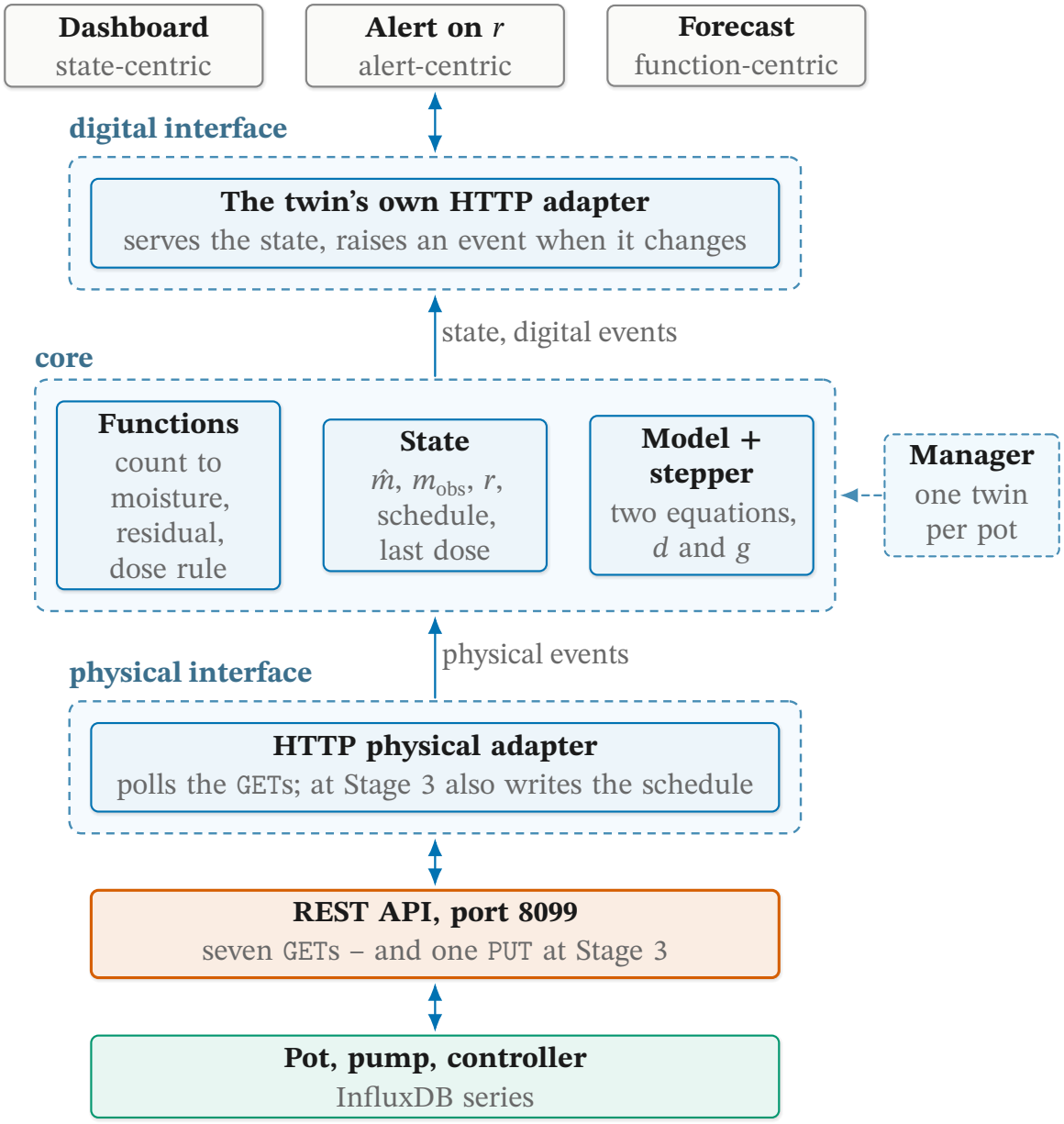


Figure 5: The rig's twin as three layers. The controller's REST API is where the physical adapter begins; the core holds the state, the two equations and the stepper; the twin's own HTTP adapter serves the state to a dashboard, an alert and a forecast. At Stage 3 nothing is added to the layers – the physical adapter gains the one `PUT`.

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.

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 [1].

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.

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
```

```yaml
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 [7]. 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.

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.

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 [38], 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.

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.

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.

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.

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.

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.

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.

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.

Further Reading

- *The five reference models, first-hand:* [29] for the five-dimension model and [30] for the software platform architecture the same group built on it; [19] for the analysts' walk through ISO 23247's entities and sub-entities, with [31] for the standard applied to use cases and [11] for how well published manufacturing architectures align with it; [32], [33], [34] and [35] for the Digital Twin Consortium's platform stack, its summary, its glossary and its systems-of-systems interoperability framework; [2] for WLDT; [38] for the six-dimension model and its interface controller.
- *Architecture studies:* [5] for the systematic mapping of digital-twin software architectures, their quality attributes and their patterns – the source of the "layered and service-oriented" finding; [25] for the pattern catalogue; [26] for the Industrial Internet Reference Architecture's "digital twin core as a middleware" pattern; [28] for the unifying reference model that maps four vocabularies onto common concepts.
- *The WLDT and Web of Digital Twins line of work:* [9] for the Web of Digital Twins, its model of properties, relationships and events, its core and augmented state and its lifecycle; [27] for the edge digital twin and its three parts; [3] for the five-step shadowing pipeline and the microservice and serverless realisations of it; [24] for the entanglement-aware lifecycle and the middleware that manages many twins; [1] for the twin's software lifecycle set against the physical twin's phases; [6] for the three layers from a security viewpoint and metric-based observability; [12] for hierarchical composition and relationship navigability; [43] and [37] for ecosystems of twins across platforms.
- *Configuration:* [17] for the open-source frameworks survey and the Eclipse Ditto workflow, with [40] and [14] for a framework built on the Ditto Thing; [41] for the comparison of digital twin description formats; [42] for DTDL's twin graph; [7] for the Digital Twin as a Service chapter – the configuration template, the minimal configuration, classes and instances, the lifecycle manager – with [44] for the case studies that fill the template in; [45] for GitOps applied to the twin lifecycle.

- *The reusable-asset and platform view:* [7] again, for assets and composition; [8] for the twin as an assembly of services built up from models and data, and [20] for the advanced services – what-if, fault diagnosis, predictive maintenance, reconfiguration; [13] for a DT Manager administering FMU models; [15] and [16] for co-simulation and its standard interface; [22] for the twin-as-a-service platform survey and its view of monitoring twin instances; [23] for a survey that places WLDT and Ditto among the platforms.
- *Consulted but not drawn on:* [39] for the Asset Administration Shell as a hierarchy of submodels, which belongs to the standards chapter; [10] for how data reaches a twin, which belongs to the data chapters; [46] for the greenhouse exemplar's architecture, cited once above as the rig's closest relative; [4] for the evolution of twins and their models over time, which the credibility chapter needs more than this one does.

References

[1] M. Picone *et al.*, "Harmonizing Physical and Digital Twins Lifecycles," in *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*, Mar. 2025, pp. 197–204. doi: 10.1109/ICSA-C65153.2025.00039.

[2] M. Picone, M. Mamei, and F. Zambonelli, "WLDT: A general purpose library to build IoT digital twins," *SoftwareX*, vol. 13, p. 100661, Jan. 2021, doi: 10.1016/j.softx.2021.100661.

[3] 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, doi: 10.1016/j.future.2024.03.052.

[4] T. Alskaif *et al.*, "Evolution at the Core of Digital Twin Engineering," Grand Rapids, USA: IEEE, Jul. 2025. doi: 10.5283/epub.77656.

[5] E. Ferko, A. Bucaioni, and M. Behnam, "Architecting Digital Twins," *IEEE Access*, vol. 10, pp. 50335–50350, 2022, doi: 10.1109/ACCESS.2022.3172964.

[6] 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. Accessed: Mar. 27, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11366535/

[7] 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. doi: 10.1007/978-3-031-66719-0_11.

[8] 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. doi: 10.1007/978-3-031-66719-0_3.

[9] 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, doi: 10.1145/3507909.

[10] 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. doi: 10.1007/978-3-031-66719-0_7.

[11] 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. doi: 10.1109/ICSA56044.2023.00015.

[12] 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. doi: 10.1109/SEAA64295.2024.00018.

[13] 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, doi: 10.1177/00375497241261406.

[14] S. Infante *et al.*, "Integrating FMI and ML/AI models on the open-source digital twin framework OpenTwins," *Software Practice and Experience*, Mar. 2024, doi: 10.1002/spe.3322.

[15] G. Schweiger *et al.*, "An empirical survey on co-simulation: Promising standards, challenges and research needs," *Simulation Modelling Practice and Theory*, vol. 95, pp. 148–163, Sep. 2019, doi: 10.1016/j.simpat.2019.05.001.

[16] 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, doi: 10.1016/j.oceaneng.2024.117675.

[17] 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, doi: 10.1002/spe.3305.

[18] 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. Accessed: Nov. 10, 2025. [Online]. Available: https://hal.science/hal-05326255/

[19] 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. doi: 10.1115/MSEC2023-101127.

[20] 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. doi: 10.1007/978-3-031-66719-0_10.

[21] 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. doi: 10.1007/978-3-031-66719-0_8.

[22] K. Duran *et al.*, "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, doi: 10.1109/COMST.2025.3635582.

[23] 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, doi: 10.1016/j.comnet.2024.110350.

[24] 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, doi: 10.1145/3699520.

[25] B. Tekinerdogan and C. Verdouw, "Systems Architecture Design Pattern Catalog for Developing Digital Twins," *Sensors*, vol. 20, no. 18, p. 5103, Jan. 2020, doi: 10.3390/s20185103.

[26] "The Industrial Internet Reference Architecture," *Industry IoT Consortium*. Accessed: Jan. 08, 2025. [Online]. Available: https://www.iiconsortium.org/iira/

[27] M. Picone, M. Mamei, and F. Zambonelli, "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, doi: 10.1145/3573206.

[28] J. Pfeiffer *et al.*, "Towards a Unifying Reference Model for Digital Twins of Cyber-Physical Systems." arXiv, Jul. 2025. doi: 10.48550/arXiv.2507.04871.

[29] F. Tao *et al.*, "Five-dimension digital twin model and its ten applications," *Comput. Integr. Manuf. Syst*, vol. 25, no. 1, pp. 1–18, 2019.

[30] F. Tao *et al.*, "makeTwin: A reference architecture for digital twin software platform," *Chinese Journal of Aeronautics*, vol. 37, no. 1, pp. 1–18, Jan. 2024, doi: 10.1016/j.cja.2023.05.002.

[31] G. Shao, "Use Case Scenarios for Digital Twin Implementation Based on ISO 23247," *NIST*, May 2021, Accessed: Mar. 06, 2024. [Online]. Available: https://www.nist.gov/publications/use-case-scenarios-digital-twin-implementation-based-iso-23247

[32] L. Heaton, "Platform Stack Architectural Framework: An Introductory Guide."

[33] kayla, "Digital Twin Platform Stack Architectural Framework by Digital Twin Consortium." Nov. 2023. Accessed: May 27, 2024. [Online]. Available: https://www.digitaltwinconsortium.org/2023/11/understanding-dtcs-digital-twin-platform-stack-architectural-framework/

[34] "Glossary of Digital Twins by Digital Twin Consortium," *Digital Twin Consortium*. Accessed: May 27, 2024. [Online]. Available: https://www.digitaltwinconsortium.org/glossary/glossary/

[35] A. Budiardjo and D. Migliori, "Digital twin system interoperability framework," Tech. rep. Digital Twin Consortium, East Lansing, Michigan, 2021. Accessed: Feb. 25, 2025. [Online]. Available: https://www.digitaltwinconsortium.org/pdf/Digital-Twin-System-Interoperability-Framework-12072021.pdf

[36] 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. doi: 10.1007/978-3-031-67778-6_5.

[37] 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, doi: 10.1016/j.softx.2025.102275.

[38] N. Li *et al.*, "Six-dimensional digital twin modeling and software platform design for complex industrial systems," *Journal of Intelligent Manufacturing*, Jan. 2025, doi: 10.1007/s10845-025-02567-8.

[39] J. Zhang, C. Ellwein, M. Heithoff, J. Michael, and A. Wortmann, “Digital twin and the asset administration shell,” 2024, Accessed: Apr. 09, 2025. [Online]. Available: https://awortmann.github.io/downloads/paper/Digital_twin_and_the_asset_administration_shell.pdf

[40] 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, doi: 10.1016/j.compind.2023.104007.

[41] 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. 200595–200608, 2025, doi: 10.1109/ACCESS.2025.3633736.

[42] 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*, in MODELS Companion ’24. New York, NY, USA: Association for Computing Machinery, Oct. 2024, pp. 332–337. doi: 10.1145/3652620.3688245.

[43] 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*, in MODELS Companion ’24. New York, NY, USA: Association for Computing Machinery, Oct. 2024, pp. 476–485. doi: 10.1145/3652620.3688263.

[44] B. J. Oakes *et al.*, “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. doi: 10.1007/978-3-031-66719-0_12.

[45] 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. Accessed: Oct. 22, 2025. [Online]. Available: https://hal.science/hal-05225783/

[46] E. Kamburjan *et al.*, “GreenhouseDT: An Exemplar for Digital Twins,” in *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, in SEAMS ’24. New York, NY, USA: Association for Computing Machinery, Jun. 2024, pp. 175–181. doi: 10.1145/3643915.3644108.