Chapter 11 – Twin Services: Visualization, Monitoring, Prediction, and Decision Support

11.0 Before you start

Where we are. Chapter 9 built the connector, Chapter 10 built the store, and both stopped at their own boundary. This chapter builds the things the rest of the world actually touches. More chapters defer to this one than to any other in Part III: Chapter 2 for visualisation and monitoring, Chapter 5 for the real-time factors it derived, Chapter 6 for optimisation, Chapter 7 for monitoring-as-a-service and for turning its threshold into a running thing, Chapter 8 for the six jobs of its Sec. 8.3, Chapter 9 for the alerting its detectors feed, and Chapter 10 for the query API.

And one debt is unlike the others. **Chapter 10 Sec. 10.5.3 found a bug and left it for this chapter to fix.** It is the only place in the book where one chapter repairs a defect another chapter demonstrated, and it is the spine of Sec. 11.4.

A note on the chapter's title, so nobody arriving from the contents page is confused. The table of contents names four services – visualization, monitoring, prediction, decision support. Chapter 3 Sec. 3.2.6 names five, and called the correspondence with Chapter 1's five value patterns "deliberately one-to-one". This chapter follows Chapter 3, because Chapter 3's component grid is what a design review checks against:

Contents page	This chapter	Chapter 1's pattern
Monitoring	Monitor (Sec. 11.2)	Monitor
(inside monitoring)	**Diagnose (Sec. 11.4)**	Diagnose
Prediction	Predict (Sec. 11.5)	Predict
Decision support	Decide (Sec. 11.6)	Decide
(not named)	Certify-and-train (Sec. 11.7)	Certify-and-train
Visualization	Visualisation (Sec. 11.3), a cross-cutting concern	*(a view onto any of them)*

Diagnose gets the most space of the five, because it is the demonstrator's only funded service and because it is where the interesting failure lives.

The register, unchanged from Chapters 9 and 10. You have built services, designed APIs, and been paged by your own alerts at three in the morning.

> **This chapter does not teach service design. It teaches what a twin's services must do that an ordinary application's services do not – which comes down to consuming data that has an age, a quality and a provenance, and producing output that a human will act on physically.**

What you are assumed to know. Everything so far. Especially: Chapter 1's five value patterns and its value metric; Chapter 2's data-flow test and its two negatives about dashboards and 3D models; Chapter 3's component grid and the services table; Chapter 5's real-time factor; Chapter 6's Monte Carlo sizing; Chapter 7's threshold, its 16 per cent

detection floor, and its monitors; Chapter 8's miss-rate and false-alarm-rate discipline and its alarm-fatigue point; Chapter 9's six quality states and completeness rule; and Chapter 10's `as_of`, `decision_time`, `inputs_watermark`, and the defect of Sec. 10.5.3.

The maths budget. As Chapters 9 and 10: more arithmetic welcome, no new mathematics. One substantial set piece – the alert economics of Sec. 11.4.5 – and it is multiplication and division throughout.

What this chapter deliberately does not cover. Service architecture, API technology and deployment – Chapter 12. Visualisation technology and rendering – Chapter 12; this chapter covers what visualisation is *for*. Human factors and interface personalisation – out; alarm fatigue appears as a design constraint, not as a literature. Natural-language interfaces – Chapter 8 Sec. 8.3.6 covered them and set the actuation-guard test, and this chapter does not reopen it. The command path's mechanism – Chapter 3 Sec. 3.2.7. Fault injection and replay as practices – Chapter 14. Optimisation algorithms – named, not taught.

Learning objectives

By the end of this chapter, you will be able to:

1. **Map** each of Chapter 1's five value patterns to a service and **determine**, from Chapter 3's grid, which components a proposed service requires and therefore what it costs.
2. **Specify** a diagnose service with **three** outcomes rather than two, and **explain** why a two-outcome service reports infrastructure failures as physical faults.
3. **Compute** a service's expected annual alert volume from measured detection and false-alarm rates, and **evaluate** a suppression strategy against the detection latency it costs.
4. **State** what a forecast service must carry beyond the forecast, and **distinguish** scenario comparison from optimisation from a learned policy.
5. **Decide** which services a twin's value case funds, and **justify** declining the rest.

11.1 What a twin service is

11.1.1 Chapter 3's table, restated as a build order

Chapter 3 Sec. 3.2.6 mapped patterns to services and Sec. 3.4 mapped services to components. Read together they are a build order, because each row needs everything the rows above it need and a little more:

Service	Adds, over the row above	Can it write to the physical twin?
Monitor	nothing – connector and store only	No
Diagnose	a model, and a state estimator	No
Predict	the simulation runner	No

Service	Adds, over the row above	Can it write to the physical twin?
Decide	the runner **at scale**, and possibly a command path	Only in its acting form
Certify-and-train	synthetic inputs; may need no physical twin at all	No

Figure 11.3 draws that table as the build order it is, with the cost of each step marked and the one writing cell isolated.

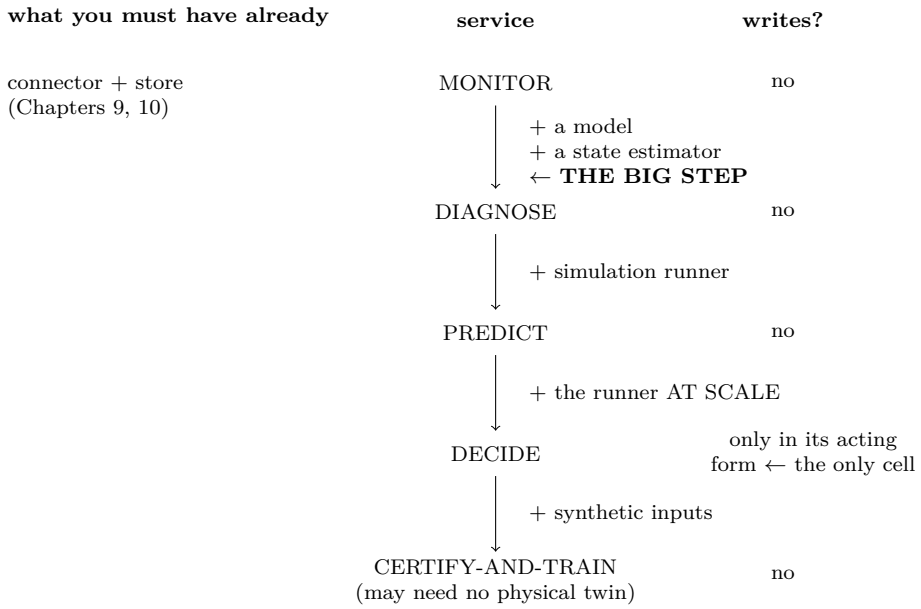

Figure 11.3 The build order. Four of the five services are deliverable with no command path at all, which is why "we need a twin" and "we need to write to the plant" are separate conversations.

Three consequences that decide project sequencing.

Monitor is nearly free once Chapters 9 and 10 exist. It needs no model. If your twin has a connector and a store, you can ship a monitor view this week, and Sec. 11.2 argues you should.

The jump from monitor to diagnose is the largest in the table, and it is where the value is. Chapter 3 said so, and Chapter 1's whole worked example depends on it.

Only one cell in the grid writes to the physical world. Four of the five patterns are deliverable with no command path at all. When somebody proposes a command path for a monitor service, Chapter 3's answer stands: that is a scope question, not a design question.

11.1.2 Four properties every twin service needs

Here is the part that is not ordinary service engineering. A twin's services consume data that has an age, a quality and a provenance, and they produce output somebody will

act on physically. Four properties follow, and a service missing any of them will behave correctly in testing and wrongly in service.

1. Age awareness. Every service must know how old its inputs are and must have a defined behaviour when they are too old. Chapter 2 Sec. 2.7.1 asked "how stale can the digital state be before the decision it feeds becomes wrong?" – that number is a service-level parameter, not a philosophical question, and Chapter 9 Sec. 9.5.4 made age of twin computable per binding.

2. Quality awareness. Chapter 9's six quality states exist so that services can behave differently. Chapter 9 Sec. 9.6.2 gave the table; this chapter is where the columns of that table become code. A service that selects `WHERE value IS NOT NULL` has thrown the distinction away at the first query.

3. Provenance emission. Every output a service produces must carry what produced it. Chapter 10 Sec. 10.5.2 defined the fields – `as_of`, `decision_time`, `inputs_watermark`, model version, parameter set – and Chapter 10 Sec. 10.6.2 showed that Chapter 7's five incident questions are each one join away *if the services wrote those fields*. They are the services' job to populate honestly, and nothing downstream can check them.

4. A defined behaviour when the input is missing. Not an exception, not a null, not the last known value: a stated behaviour with a name. This is Sec. 11.4.2's third outcome, and it is the chapter's central idea.

> **A test you can apply to any twin service in a design review.** Ask what it does when its newest input is two hours old, when that input is marked `substituted`, and when it did not arrive at all. If the three answers are the same answer, the service has one of the four properties and needs four.

11.1.3 Who may write, restated once

Chapter 2 Sec. 2.8 established that closing the loop changes the obligations, and Chapter 3 Sec. 3.2.7 built the actuation guard. The service-layer restatement is one sentence: **exactly one service may reach the command path, its writes go through the guard, and every command is audited with the state that justified it.** Sec. 11.6.4 returns to it. Everything else in this chapter writes to screens, queues and tables.

11.2 Monitor: the cheapest pattern, and not nothing

11.2.1 What it is

Current and historical state, presented. No model, no estimator, no runner. Chapter 2 Sec. 2.6.2 classified it precisely and without insult: **a monitoring dashboard is a digital shadow, not a digital twin** – and shadows earn money, which is why Chapter 1's nine-month payback case was one.

Build it first, and not only because it is cheap. A monitor view is the first thing that will tell you your connector is wrong. Chapter 9's completeness counts and age-of-twin metric are numbers; a plot of the last week with gaps visible is the same information in a form where a human spots the problem in two seconds. **Every twin project should**

have a monitor view before it has a model, because it is the cheapest possible test of Chapters 9 and 10.

11.2.2 Three things a monitor view must show that a dashboard usually does not

A general-purpose dashboard plots values against time. A twin's monitor view needs three more things, and each corresponds to something an earlier chapter fought for.

Gaps as gaps. Chapter 9 Sec. 9.6.1 created `no_reading` rows so that absence is a fact rather than a hole. A plotting library will happily draw a straight line between the readings on either side of a two-hour outage, and that line is a lie the eye believes. **Break the line.** This is the single highest-value plotting decision in a twin, and it is one configuration option in most libraries.

Quality, distinguishably. A `substituted` value should not look like a measured one. A hollow marker, a lighter colour, anything – but different. Chapter 10 Sec. 10.3.3 made the distinction storable; here it becomes visible.

Measured and estimated in different colours, always. Chapter 5 Sec. 5.2.4 warned that a twin storing predicted and measured in one table with a flag will eventually plot one as the other. The plotting layer is where "eventually" happens.

11.2.3 Age of twin belongs on the screen

Chapter 9 Sec. 9.5.4 called age of twin the best single health metric a twin has, because it responds to a dead sensor, a dead gateway, a dead network, a crashed connector and a full disk. Put it on the monitor view, per binding, as a number and not as a chart.

And the reason it must be on the same screen as the data, rather than on an operations dashboard somebody else looks at: the person reading the moisture plot is the person who needs to know that the plot stops being current at 14:20. A stale chart with a fresh-looking axis is how a human comes to trust a number that is six hours old.

11.3 Visualisation, which is a view and not a twin

Chapter 2 Sec. 2.6.1 promised this section: "Presentation is a service the twin offers, and Chapter 11 covers it properly. It is not the twin."

11.3.1 What visualisation is for

The purpose is not decoration and it is not marketing. Its object is to lower the effort a human spends getting wanted information out of data, and in a twin specifically to let users draw the most from the models, data and analytic services on offer – showing the system's state and dynamics together with what simulation, analysis and prediction add to them [1]. Two consequences of that framing are worth stating because they are load-bearing.

It widens the audience. Visualisation lets people with a modest system-level understanding use and act on what the twin knows, taking the audience past the data engineer

and out to field staff who have no technical background [1]. That is a real business function, not a nicety.

It is bounded by dimensionality, not by taste. In principle one might want to consider ten variables over time; in practice it is hard to visualise more than three dimensions plus time [1]. So the design question is always *which three* – and answering it requires knowing the decision, which is Chapter 1 again.

11.3.2 The delete-the-geometry test, and when geometry earns its place

Chapter 2 gave the test: ask what happens if you delete the geometry. If the twin still answers its questions, the geometry was a view. If the whole thing stops, you had a Computer-Aided Design (CAD) model with a data feed.

That test is a sorting rule, not a prohibition. Geometry earns its place in exactly three situations, and it is worth knowing them so you can say yes for a reason:

1. **The answer is spatial.** Where is the hot spot, which joint is cracking, which aisle is congested. A number cannot carry a location and a plot cannot carry a shape.
2. **The audience is spatial.** A technician who will walk to the thing navigates by where it is. "Pot 3, bay 2" is a map problem.
3. **The model is spatial.** Chapter 6 Sec. 6.3.4 covered finite elements; a model whose state is distributed over a geometry has results that are only meaningful on that geometry.

For the demonstrator, all three are false, and the honest visualisation is a plot. Twelve pots on a bench do not need a rendered greenhouse, and building one would be Chapter 1's unpaid work in its purest form. Platform comparisons note that twins present data as 3D models and interactive dashboards to help stakeholders follow complex processes [2] – true, and the question is always whether *this* stakeholder following *this* process needs it.

11.3.3 Four audiences, four views

The commonest visualisation failure is one view for everybody. Four audiences, and their needs barely overlap:

Audience	Wants	Time horizon	The mistake
The technician acting now	What is wrong, where, and what to do	Now	Giving them the residual plot
The engineer investigating	Measured against predicted, with quality and gaps	Days	Giving them a summary
The scientist or operator planning	Trends and scenario comparisons	Weeks	Giving them live data at 10-minute resolution
The sponsor	The value metric of Chapter 1, in its own units	Quarters	Giving them anything technical at all

The fourth row is the one engineers get wrong. Chapter 1's whole method was to name a value metric in the sponsor's units – experiment-weeks lost to undetected watering faults – and the sponsor's view should show that number and its trend, not moisture. A twin that cannot report its own value metric will have its budget questioned by someone who has no way to answer the question.

11.3.4 The plot that actually gets used

For the demonstrator, one plot does most of the work, and it is worth describing because its composition is the chapter in miniature:

> Measured moisture, as points, with gaps broken and substituted values marked differently. The model's expected trajectory over the same window, as a line in a different colour. Commanded watering events as vertical marks with their doses. Alerts as annotations. And the age of twin, as a number in the corner.

Five layers, four of which exist only because Chapters 9 and 10 kept something a simpler pipeline would have discarded. **A technician looking at that plot can diagnose most of what goes wrong with this twin without reading anything else**, which is a better description of a successful visualisation than any statement about fidelity.

11.4 Diagnose: the service that finds the fault

This is the demonstrator's only funded service (Chapter 3 Sec. 3.7.6), the one Chapter 1 built its value case on, and the one with the interesting failure.

11.4.1 The loop

For each commanded watering event: work out what the model expected, work out what was observed, compare, and act on the comparison. Chapter 7 Sec. 7.5.2 supplied the numbers – expected step 56.4 reading units for a 120 ml dose, alert when the observed step falls more than 9.0 units below that.

Naively, then:

```
on each watering event e:
    expected = g * (e.dose / 100)
    observed = reading_after(e) - reading_before(e)
    if observed < expected - 9.0:  raise fault alert
    else:                          all clear
```

That is the service Chapter 10 caught being wrong, and everything below is the repair.

11.4.2 Three outcomes, not two

Here is the chapter's central idea, and it generalises far beyond this service.

> **A service that can only answer "problem" or "no problem" will report infrastructure failures as physical faults.**

The loop above has two outcomes. But there is a third state of the world it cannot express: *the inputs needed to decide did not arrive.* When the post-dose reading is missing, the service does not know whether the dose landed. It knows that it does not know, and "I do not know" is a different answer from "the pump failed" – different cause, different owner, different action, different cost.

Outcome	Means	Goes to	Action
`fault`	The dose demonstrably did not deliver	Duty researcher	Walk to the rig
`healthy`	The dose demonstrably delivered	Nobody	None
`cannot_evaluate`	**The data needed to decide is absent or unusable**	**Whoever owns the connector**	**Fix the pipeline**

Figure 11.1 shows where the third outcome comes from, and why a two-outcome service does not merely lose information – it actively misroutes.

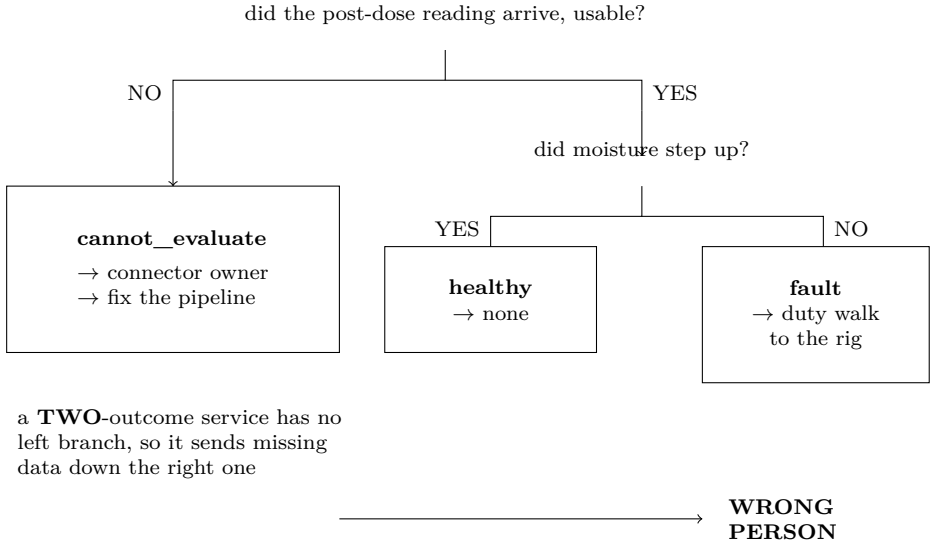

Figure 11.1 Three outcomes. The left branch is the one Chapter 9's six quality states were built to make possible; a service without it converts every infrastructure failure into a physical-fault alert.

Notice that Chapter 9 built the vocabulary for this two chapters ago and nothing consumed it. The six quality states exist precisely so that a service can tell "no water arrived" from "no reading arrived". A service with two outcomes throws that distinction away at the last step, after two chapters of work to preserve it – which is a good description of how carefully-built infrastructure comes to nothing.

And the third outcome is the one that changes the alert economics, as Sec. 11.4.5 computes: it is more common than the fault it is confused with.

11.4.3 Fixing Chapter 10's defect

Chapter 10 Sec. 10.5.3 walked an incident: on 14 August the service alerted "dose did not land" at 17:22, and the dose had in fact landed. Chapter 10 found two defects and named the fix as this chapter's. Here it is.

Defect 1: the service subtracted without checking its operands. It took the newest available reading as the post-dose value without checking that it was actually *after* the dose. Both operands were the 17:00 reading, so the step came out as 0 – an arithmetic artifact of subtracting a reading from itself, presented as a measured absence of water.

Defect 2: the service evaluated before its data could have arrived. It ran at 17:22 against an input watermark of 17:00. Chapter 9 Sec. 9.7.3 allowed one poll interval of lateness; the service ignored the allowance.

The repaired loop, with every added line traced to its source:

```
on each watering event e:
    before = last_good_reading(e.binding, before=e.time)
    after  = first_good_reading(e.binding, after=e.time, within=window)

    # Defect 1: the post-dose operand must exist and be after the dose.
    if after is None:
        if now() < e.time + window + lateness_allowance:
            defer            # Ch9 Sec. 9.7.3: late data may still arrive
        else:
            return cannot_evaluate("no post-dose reading")
    if before is None:
        return cannot_evaluate("no pre-dose reading")

    # Ch9 Sec. 9.6.2: quality decides admissibility, not nullness.
    if before.quality != good or after.quality != good:
        return cannot_evaluate("post-dose reading not usable: " + after.quality)

    expected = g * (e.dose / 100)
    observed = after.value - before.value
    verdict  = fault if observed < expected - threshold else healthy

    # Ch10 Sec. 10.5.2: emit provenance or the incident review is impossible.
    return verdict with (as_of=after.event_time,
                         decision_time=now(),
                         inputs_watermark=newest_event_time_visible(),
                         model_version, parameter_set_version,
                         expected, observed)
```

Re-run 14 August against it. At 17:22 there is no reading after 17:05, and the lateness allowance has not expired, so the service **defers**. It runs again after the allowance; by then the 18:30 backfill has delivered the 17:10 reading of 663, so `observed` = 663 - 606 = 57.0, which is within one spread of the expected 56.4, and the verdict is `healthy`. **No alert, no incident, no week spent hunting a threshold bug in November.**

And if the backfill had never come, the service returns `cannot_evaluate("no post-dose reading")` – which goes to whoever owns the connector, not to a researcher who would have walked to a healthy rig.

> **Three lines of defensive checking, and they are the difference between a service that is right and a service that is right when the data is perfect.** Chapter 9 and Chapter 10 spent two chapters making the distinction between absent and zero available; this is the code that spends it.

11.4.4 The alert as an object

An alert is not a message. Chapter 3 Sec. 3.2.7 required every command to be audited with the state that justified it, and Chapter 7 Sec. 7.7.3 listed the five questions an incident review asks. An alert must carry enough to answer them:

```
alert(
    id, binding, raised_at,
    outcome,                    -- fault | cannot_evaluate
    reason,                     -- the human sentence
    dose_event_id, expected, observed, threshold,
    as_of, decision_time, inputs_watermark,
    model_version, parameter_set_version,
    state_id,                   -- the twin_state row that justified it
    acknowledged_at, acknowledged_by, resolution
)
```

Four fields carry more weight than they look.

***`threshold`** is stored, not looked up later.* Chapter 7's threshold is derived from a calibration that will be refitted. An alert that records the threshold it fired against is self-explaining; one that does not is interpreted years later against a number that has changed.

***`state_id`** points at the belief, not at a copy of it.* Chapter 10's `twin_state` row has the `decision_time` and the watermark; pointing at it keeps one source of truth.

***`resolution`** closes the loop on the value metric.* Chapter 1 measured experiment-weeks lost to undetected faults, and Sec. 11.3.3 said the sponsor wants that number. It is computable only if somebody records what each alert turned out to be. **This is the field teams skip and then cannot report their own value.**

***`acknowledged_at`** is how you measure whether the service is being ignored.* Which Sec. 11.4.5 argues is the thing to worry about.

11.4.5 Alert economics, worked

Nobody costs an alerting service before building it, and the arithmetic takes ten minutes. Here it is for the demonstrator, at twelve pots – the same hypothetical plant count Chapter 10 Sec. 10.3.5 used, and the conclusions hold at any count you like.

The inputs, all from earlier chapters:

- Dose events per year: `12 x 2 x 365 = 8,760`.

- Genuine fault episodes: Chapter 1's baseline was three in three months, so **12 a year**. With the twin deployed, a fault is found and fixed the same day, so each episode produces about **2** raw alerts before repair.
- False-alarm rate: Chapter 8 Sec. 8.5.1 measured the physics detector at **2 false alarms in 194 healthy doses**, or 1.03 per cent.
- Incomplete windows: **2 per cent**, meaning one dose window in fifty is unusable at evaluation time by Chapter 9's completeness rule. **This is the one number here with no provenance** – the other three come from Chapter 1, Chapter 8 and arithmetic. It is illustrative, in the sense Chapter 9 Sec. 9.0 used for datasheet figures: the method transfers, the value does not, and it is measurable on your own connector in a week. It also happens to be the input the chapter's largest finding rests on, so measure it before quoting that finding.

The arithmetic:

```
raw fault alerts        = 12 episodes x 2      =  24
false alarms            = 8,736 x 0.0103       =  90
cannot-evaluate alerts  = 8,760 x 0.02         = 175
                                        total = 289 per year
                                              = about 5.6 per week
```

And the number that matters, which nobody computes:

```
signal ratio = 12 genuine episodes / 289 alerts = 1 in 24
```

One alert in twenty-four is a real fault. Chapter 8 Sec. 8.11.4's solution named the consequence: a detector firing that often will be muted, and a muted detector has a miss rate of 100 per cent regardless of what any test said. **This service, built exactly to Chapter 7's specification and with a measured 0 per cent miss rate, would fail in the field within a month.**

Notice also that the largest category is not the false alarms. It is `cannot_evaluate` – 175 of 289, more than the faults and false alarms combined. A two-outcome service (Sec. 11.4.2) reports every one of those 175 as "dose did not land", which means **60 per cent of this service's alerts would be infrastructure failures wearing a plant fault's clothes.**

11.4.6 Three fixes, and what each costs

Fix 1: route `cannot_evaluate` to a different owner, aggregated. It is not a plant problem. Send it to whoever owns the connector, as one daily digest rather than 175 individual alerts.

```
researcher's queue: 289 - 175 = 114 per year
```

Cost: nothing. This is the fix Sec. 11.4.2 was for, and it is free.

Fix 2: suppress duplicates. A pot with a failed pump fails every dose until somebody fixes it. One *open* alert per binding per condition, updated rather than re-raised.

```
researcher's queue: 114 - 12 = 102 per year, signal ratio 1 in 8.5
```

Cost: nothing, and it also makes `resolution` meaningful, because there is one alert per episode to resolve.

Fix 3: require confirmation. Alert only after **two consecutive** doses fail on the same pot.

A genuine pump failure persists, so confirmation costs detection latency and not detection. A noise-driven false alarm does not persist, so two in a row on the same pot is the product of two small rates:

```
false-alarm rate per dose      = 0.0103
probability of two in a row    = 0.0103 x 0.0103 = 0.000106
consecutive pairs per year     = 12 pots x 729   = 8,748
both doses evaluable (0.98^2)  = 8,748 x 0.96    = about 8,400 usable pairs
expected confirmed false alarms = 8,400 x 0.000106 = about 0.9 per year
```

Two doses in a row is not two scheduled doses in a row, and the difference has to be decided rather than assumed. Sec. 11.4.2's third outcome means a dose between two others may be neither pass nor fail: `cannot_evaluate` neither confirms a pending detection nor clears it. So "consecutive" must mean **the next dose that could be evaluated**, which is why only about 96 per cent of adjacent pairs are usable above. Three rules follow, and all three belong in the service's specification:

1. **A pending detection survives an unevaluable dose** and waits for the next evaluable one. Dropping it would silently discard a real detection; confirming on it would defeat the purpose.
2. **The wait is bounded.** A pending detection unconfirmed after 48 hours is escalated as an *unconfirmed suspicion* to the researcher, with a `cannot_evaluate` to the connector owner. An unbounded pending state is how a fault goes unreported while the system believes it is being careful.
3. **The latency claim must account for it.** About 102 first detections a year (12 genuine plus 90 false), of which 2 per cent meet an unevaluable confirming dose – so **about twice a year a detection waits an extra dose**, taking the worst case to roughly 24 hours instead of 16. Almost all of those two are false first detections that then clear.

```
researcher's queue: 12 + 0.9 = about 13 per year, one every four weeks
signal ratio: 12 in 13
```

From one alert in twenty-four to twelve in thirteen.

What fix 3 costs, stated precisely. Detection waits for the next dose. Doses are at 09:16 and 17:05, so the delay is about 8 hours if the morning dose fails and about 16 if the evening one does. Check that against the requirement rather than against instinct: Chapter 1's baseline was faults undetected for **four days**, and Chapter 2 Sec. 2.7.1 said a state one hour old is fine and one week old is useless. **Sixteen hours against a ninety-six-hour baseline is affordable**, and so is the twice-a-year twenty-four, and this is the check that distinguishes an engineering decision from a preference.

11.4.7 What confirmation does not fix, and it matters

The multiplication in fix 3 assumes the two false alarms are **independent** – that a spurious low reading on one dose tells you nothing about the next, eight hours later. For false alarms driven by sensor noise, that is reasonable.

It is false for anything systematic. A sensor drifting low, a dripper partially blocked, a calibration that has gone stale – these produce *every* dose reading low, so confirmation confirms them enthusiastically and the false alarms survive at full rate. This is Chapter 7's distinction between bias and spread arriving in the alerting layer: **confirmation defeats spread and does nothing to bias.**

Which tells you what else has to exist. Chapter 7 Sec. 7.7.4's monitors – rolling residual bias, rolling residual spread, estimator gain – are the things that catch the systematic case, and Sec. 11.8.4 makes them services. **Confirmation and drift monitoring are not alternatives; each is blind to what the other catches**, and a service that has only one of them has a known gap that should be in Chapter 7's credibility argument.

11.4.8 What to ask

- What are this service's three outcomes, and where does each one go?
- What does it do when the reading it needs has not arrived yet, as opposed to definitely not coming?
- How many alerts a year, and what fraction of them will be real?
- Who acknowledges them, and what is measured about whether they do?
- Which false alarms does the suppression strategy *not* remove?

11.5 Predict: the forecast service

11.5.1 What a forecast owes beyond the forecast

A forecast is not a number. Four things must travel with it, and each is owed to an earlier chapter.

A horizon and an `as_of`. "Moisture will be 606" is meaningless. "Moisture at 17:05, forecast as of 09:20" is a claim.

A range. Chapter 7 Sec. 7.6.3 established the demonstrator's day-ahead prediction as 606.6 plus or minus 2.0, and was careful that the range supersedes a component-by-component budget rather than adding to it. **A forecast service that emits a bare number has discarded the entire content of Chapter 7.**

The assumptions about inputs. A twelve-hour forecast depends on what happens in those twelve hours – the scheduled doses, the weather. Chapter 7 Sec. 7.6.1 listed input and scenario uncertainty as the one source that cannot be reduced by better modelling. So the forecast must say which scenario it assumed, and a forecast under an assumed schedule that then changes is not wrong, it is answering a different question.

Its provenance. Chapter 10 Sec. 10.5.2's fields, exactly as for diagnose.

11.5.2 Where the range comes from, and where it does not

Two sources, and confusing them is the standard error.

From Chapter 7's hold-out. The model-reality gap, measured end to end. This is the honest default and it is what Sec. 11.5.1's "plus or minus 2.0" is.

From Chapter 6's Monte Carlo. Propagating stated input uncertainty through many runs. This is more expensive and answers a different question – how much does the *answer* move when the *inputs* move – and Chapter 6 Sec. 6.7.4 and Chapter 7 Sec. 7.6.2 were both explicit that it propagates stated uncertainty and cannot discover unstated uncertainty.

Use the hold-out range for a forecast under a known scenario, and Monte Carlo when the scenario itself is uncertain. Reporting both, added together, is Chapter 7 Sec. 7.6.3's double-counting trap.

11.5.3 The real-time factor as a request budget

Chapter 5 Sec. 5.3.2 derived which patterns need a real-time factor above one, and this chapter owes only the service-level consequence, which is short.

Predict needs the forecast before the horizon elapses – a twelve-hour forecast computed in thirteen hours is not a forecast. For the demonstrator this is not a constraint: Chapter 5 Sec. 5.3.4 costed a whole run at 32 microseconds.

The real constraint is the request, not the horizon. If the forecast is computed on demand behind an API, Chapter 5's arithmetic must fit inside whatever timeout sits in front of it – and Chapter 6 Sec. 6.3.3's warning about adaptive solvers applies: a solver that chooses its own step size has a tail-latency problem that will not appear in testing on quiet data. **Either bound the solver's work or compute forecasts on a schedule and serve them from the store.** For most twins the second is right, and it also gives you Sec. 11.5.4 for free.

11.5.4 The scoring loop, which almost nobody builds

Here is a service that costs a day and pays for years: **compare each past forecast against what actually happened, continuously.**

You already have everything. Chapter 10 retains derived output with its `as_of` and `decision_time`, so yesterday's twelve-hour forecast is in the store next to today's measurements. Joining them gives a residual per forecast per horizon.

Figure 11.2 is the loop, and the reason it costs a day is that every arrow already exists.

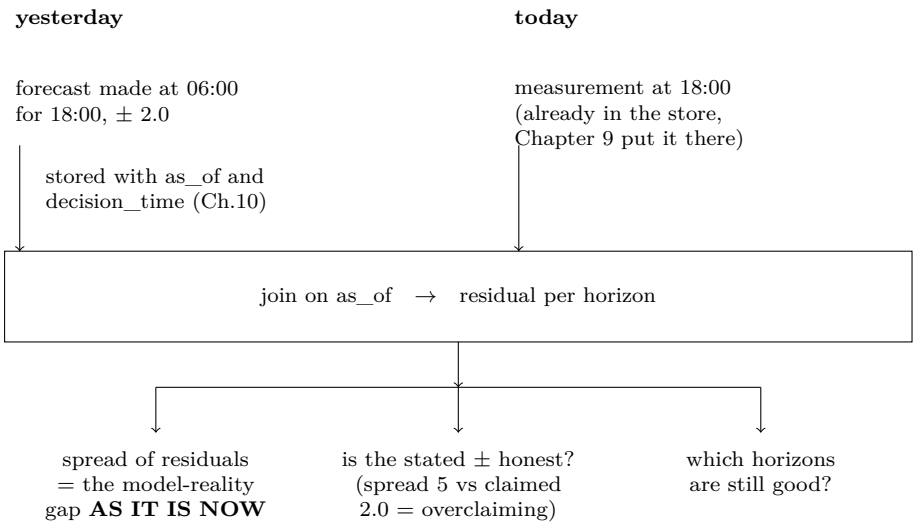

Figure 11.2 The scoring loop. No new storage, no new instrumentation: Chapter 10 kept the forecasts and Chapter 9 kept the measurements, and the join is what nobody builds.

What it buys, and each item is something an earlier chapter wanted:

- It is Chapter 7's hold-out, running forever, on live data. The spread of scoring residuals is the model-reality gap **as it is now**, not as it was in week 3.
- It gives Chapter 7 Sec. 7.7.4's expiry monitors their input, for the forecast service specifically rather than for the estimator.
- It makes the forecast's stated range checkable. If you publish plus or minus 2.0 and the scoring residuals have a spread of 5, you are overclaiming, and the service can notice this about itself.
- It distinguishes horizons. A forecast good at two hours and poor at twelve is a normal and useful thing to know, and it is invisible without scoring.

Score by horizon, not in aggregate. Averaging a two-hour forecast's error with a twelve-hour forecast's error produces a number that describes neither.

11.5.5 Prognostics, briefly

The industrial form of prediction is remaining useful life, and it is the largest commercial application of twins that involves forecasting. It has its own systematic literature [3], its own reference architectures [4], and its own overviews [5]. Chapter 8 Sec. 8.3.2 covered what makes it hard – labels for "time until failure" require failures, you get one per asset per lifetime, and the assets you most want to predict have not failed.

The service-level consequence, which is the part this chapter owes: a remaining-life output must carry its range even more insistently than a short-horizon forecast, because the decision it feeds – send a vessel, extend a certification – is expensive and irreversible. A point estimate of remaining life is not a forecast, it is an invitation.

11.6 Decide: scenario comparison and optimisation

11.6.1 What-if as a service

Chapter 3 called this row *decide*, and its first form is not a decision at all: it is presenting alternatives to somebody who will decide.

> A what-if simulation evaluates the behaviour of the physical twin under different circumstances and hypotheses; where unwanted behaviour is detected, it lets a human operator or the twin itself compare interventions and inspect the consequences before applying them, so a more informed decision can be made [6].

Three service-level requirements follow from that sentence, and they are what separates a what-if service from a script somebody runs.

The scenarios must be reproducible. Chapter 5 Sec. 5.6's requirement, arriving as an API contract: a what-if result must be re-runnable, which means the scenario definition is stored, not passed and forgotten.

The starting state must be pinned. Every scenario in a comparison must start from the *same* twin state, or you are comparing scenarios and starting conditions at once. Chapter 10's `as_of` is what pins it.

The comparison must be presented as a comparison. Five separate forecast results are not a decision aid. The output is a ranked table with the metric that ranks them named – which is Chapter 1's value metric again, and if nobody can name it the service has no output specification.

11.6.2 Optimisation is a search, and it is not learning

Chapter 8 Sec. 8.3.5 drew this distinction and handed the optimisation half here. Restating it because the two get called the same thing:

	Optimisation	Learned control policy
What it is	A search over candidate actions, running the model on each	A model that outputs the action directly
Where the knowledge is	In the model	In the weights
What you can inspect	The candidates, their scores, and the winner	Nothing
Latency	Model runs times candidates	Microseconds
What it needs	A model and compute	Training data and a training procedure

Optimisation involves no learning at all, and it is the right first answer for almost every twin, because its reasoning is inspectable: here are the candidates, here is the objective, here is what each scored. Chapter 8 Sec. 8.3.5's rule stands – do not make a learned policy the first thing you close a loop with.

What you do have to decide is the objective, and it is the same decision Chapter 7 Sec. 7.4.3 flagged for calibration. An optimiser will enthusiastically minimise whatever it is given. "Minimise water" produces a dead plant; "maximise moisture" produces a flooded one. The objective is a business statement and it belongs to whoever owns Chapter 1's value metric. Design-space-exploration techniques and their trade-off surfaces are the standard machinery here [6], and the machinery is not the hard part.

11.6.3 Sizing it, from Chapter 6

Chapter 6 sized these runs and this chapter only has to spend the numbers. For the demonstrator: a run costs 672 steps at about a microsecond a step, and Chapter 6 Sec. 6.7.3 established that a roughly ten per cent answer needs about 1,000 Monte Carlo runs. So a what-if service comparing five candidate schedules, each with its uncertainty propagated:

```
5 candidates x 1,000 runs x 672 microseconds = about 3.4 seconds
```

Which fits in a request, comfortably, and settles the architecture: no job queue, no asynchronous result, no polling endpoint. Chapter 5 Sec. 5.3.4's question – does it fit in a request? – answered with arithmetic rather than with an assumption, and answered *before* somebody builds the job queue.

The same arithmetic for the turbine gives a different architecture, and that is the point of doing it.

11.6.4 Advisory and acting, and the one cell in the grid

Chapter 3 Sec. 3.4's grid has exactly one cell that writes to the physical twin: decide, in its acting form. Chapter 2 Sec. 2.8 established what changes there.

The service-layer restatement, in four rules:

1. **The decision and the command are separate steps**, and the guard sits between them. A service that computes a recommendation and issues it in one function has no place to put the guard.
2. **Every command carries the state that justified it** – Chapter 3 Sec. 3.2.7 and Chapter 10's `alert` object shape, applied to commands.
3. **Fail-safe is defined per command, in advance.** What happens when the twin state is stale, when the model registry cannot resolve a version, when the connector is down. Chapter 3 listed these; the service is where they become branches.
4. **The maximum age at which acting is allowed is a number in configuration.** Chapter 3 Sec. 3.3.2 called this Chapter 2's twinning-rate question arriving as a line of code. Here it is that line.

And the honest note about the demonstrator, which Chapter 3 already made: it is at Stage 2, advisory, and Sec. 11.9 does not build a command path. Chapter 2's sentence stands – "it will tell you within eight hours that pot 7 stopped receiving water; it will not water pot 7; making it water pot 7 is a separate increment, and here is what that costs."

11.7 Certify-and-train

11.7.1 What it is, and the thing worth noticing about it

Replay, rehearsal, fault injection, operator training. Chapter 3 Sec. 3.2.6 made an observation about this row that is worth repeating because it prevents a class of confusion:

> **Certify-and-train may not need a physical twin at all.** A rehearsal environment running models against synthetic inputs is a *digital model* by Chapter 2's test. It shares components with the twin and is not one.

Saying that in a review is worth an hour, because the alternative is a long argument about whether the training environment needs a live connection to an asset that is, in the scenario being rehearsed, on fire.

What the service needs, and it is less than people expect: the models, the runner, and a source of inputs that is not the connector. Chapter 5 Sec. 5.5.3's simulate-the-physical-twin arrangement is exactly this, and Chapter 5 already built it as a testing device.

11.7.2 Named, and deferred

Fault injection is the technique that makes rehearsal systematic, and it belongs to this family of advanced twin services [6] – but it is framed there, correctly, as a **validation technique for dependability**, which makes its practice Chapter 14's rather than this chapter's. Chapter 5 Sec. 5.12 already sent replay and simulate-the-physical-twin to Chapter 14 for the same reason.

What this chapter asserts and stops: certify-and-train is a real row in Chapter 3's grid, it is the only row that may not need the physical twin, it reuses the runner and the models, and Chapter 1 Sec. 1.3's fifth pattern said it needs everything in Chapter 7 plus evidence. The demonstrator's value case funds none of it, and Sec. 11.9.5 declines it explicitly.

11.8 Four cross-cutting service concerns

11.8.1 The query API, and what it must not expose

Chapter 10 Sec. 10.0 deferred query APIs here. Three rules, each of which protects something an earlier chapter built.

Never expose a value without its quality. An API returning `{"moisture": 606}` has undone Chapter 9 Sec. 9.6 at the boundary, and every consumer will treat every number as measured. Return the quality, and make it required rather than optional in the response schema.

Never expose a belief without its `as_of`. Chapter 3 Sec. 3.3.2 required `as_of` to be an output rather than a log line. An endpoint returning current state without saying how current is the same defect as a stale chart with a fresh axis (Sec. 11.2.3).

Expose the two questions separately. Chapter 10 Sec. 10.5.1 distinguished "what was measured at t" from "what did the twin believe at t, at that moment". Those are

two endpoints, or one endpoint with a required parameter – but not one endpoint whose meaning depends on who is asking.

11.8.2 Idempotent re-evaluation

Every service in this chapter will be re-run: because late data arrived (Chapter 9 Sec. 9.7.3), because a parameter set was refitted (Chapter 7 Sec. 7.4.7), because somebody is reproducing an incident.

So make re-evaluation a normal operation, not a recovery procedure. Two requirements:

Re-evaluating produces a new row, not an edit. Chapter 10's `(binding, as_of, decision_time)` key was designed for this: the same `as_of` evaluated twice gives two rows with different `decision_time`s, and both are true. **Do not update the old belief.** The old belief is what the twin acted on.

Re-evaluation must not re-raise alerts. A service re-run over last month should not page anybody. That means the alerting side effect is separate from the evaluation, which is a small design decision to make early and an awkward one to retrofit.

11.8.3 What happens when a service is down

Chapter 8 Sec. 8.3.8 asked this of any service-side proposal – a service with no defined behaviour when down is a dependency nobody costed. It applies here with a twin-specific twist.

An ingest gap is recoverable; a missed evaluation may not be. If the connector stops for two hours, Chapter 9's backfill recovers the measurements. If the *diagnose service* stops for two hours, the measurements are safe – but the doses that happened in those two hours were never evaluated, and nothing will notice unless somebody arranged for it.

The fix is a work queue rather than a schedule. Do not evaluate "every ten minutes"; evaluate "every watering event that has no verdict and whose window has closed". Then a stopped service catches up by itself, and a service that has fallen behind is visible as a queue depth. This is the same instinct as Chapter 9's watermark, applied to work rather than to data.

11.8.4 Chapter 7's monitors, as services

Chapter 7 Sec. 7.7.4 specified three monitors and Chapter 7 Sec. 7.13 said this chapter would build them. They are the smallest services in the chapter and among the most valuable:

Monitor	Reads	Fires when	Goes to
Residual bias	Scoring residuals (Sec. 11.5.4) or diagnose residuals	14-day rolling mean outside plus or minus 1.0	Model owner: refit trigger
Residual spread	Same	Rolling spread grows materially	Connector owner: sensor or plumbing

Monitor	Reads	Fires when	Goes to
Estimator gain	The state estimator	Sustained move outside its stated band	Model owner: expiry trigger

Three notes.

These are expiry triggers, not alerts. Chapter 7 Sec. 7.7.2 made "the credibility argument has expired" a state of the world. Firing one of these should mark the argument stale, not page a researcher at 3 a.m.

They catch what confirmation cannot. Sec. 11.4.7's point: confirmation defeats spread and does nothing to bias, and the bias monitor is the thing that catches bias.

They are the reason Sec. 11.5.4's scoring loop is worth building, because scoring gives them a live input instead of a quarterly study.

11.9 Worked example: the demonstrator's services, built

Chapter 3 Sec. 3.7.6 specified this in eight lines and declined three services. Here it is in full, and the declining is part of the work.

11.9.1 What is built

Diagnose, per Sec. 11.4.3's repaired loop, with three outcomes, confirmation, suppression, and the alert object of Sec. 11.4.4.

Monitor, per Sec. 11.2: the five-layer plot of Sec. 11.3.4 and age of twin per binding.

The three expiry monitors of Sec. 11.8.4, because they cost a few lines each and Chapter 7's credibility argument depends on them.

Nothing else.

11.9.2 The diagnose service's parameters, all traced

Parameter	Value	Source
Expected step	`g x dose/100`, `g` = 47.0	Ch7 Sec. 7.4.2
Alert threshold	9.0 reading units below expected	Ch7 Sec. 7.5.2, three spreads
Evaluation window	30 minutes after the dose	Ch4 A3: water mixes fast relative to `Ts`
Lateness allowance	one poll interval, 10 minutes	Ch9 Sec. 9.7.3
Admissible quality	`good` only	Ch9 Sec. 9.6.2
Confirmation	2 consecutive **evaluable** failed doses, same binding; a pending detection survives an unevaluable dose	Sec. 11.4.6

Parameter	Value	Source
Pending-detection bound	48 hours, then escalate as unconfirmed suspicion	Sec. 11.4.6
Suppression	one open alert per binding per condition	Sec. 11.4.6
`cannot_evaluate` routing	daily digest to the connector owner	Sec. 11.4.2

11.9.3 The expected alert volume, stated up front

From Sec. 11.4.5 and Sec. 11.4.6, at twelve pots:

```
genuine fault episodes          12 per year
confirmed false alarms           1 per year
researcher's queue              13 per year, about one every four weeks
signal ratio                    12 in 13
cannot-evaluate, aggregated    175 per year, as about 365 daily digests
                               (most of them empty)
detection latency               8 to 16 hours, against a 96-hour baseline
                               (about 24 hours twice a year, when the
                                confirming dose is itself unevaluable)
```

Putting that table in the design document before building is the whole point of Sec. 11.4.5, because every number in it is a decision somebody should get to argue with.

11.9.4 What Chapter 7's credibility argument gains, and loses

Gains. The claim of Chapter 7 Sec. 7.8.5 can now state a detection latency and an alert volume, both derived rather than hoped for.

Loses – and this must be written down. Confirmation raises the detection latency from one dose to two, so Chapter 7's claim sentence changes:

> The twin's watering-fault detector raises an alert when **two consecutive evaluable** scheduled doses deliver 16 per cent less water than commanded, or less. Detection latency is 8 to 16 hours, extending to about 24 when the confirming dose cannot be evaluated; a detection unconfirmed after 48 hours is reported as an unconfirmed suspicion. **A single failed dose followed by a successful one is not reported**, and confirmation does not reduce false alarms arising from a systematic cause such as sensor drift; the residual monitors of Sec. 11.8.4 cover that case.

That is a weaker claim than Chapter 7's and a truer one, and it is the kind of amendment that only appears when somebody builds the service.

11.9.5 What is declined, and why

Declined	Why
Forecast	Chapter 1's value metric is faults detected, not moisture predicted. Nothing acts on a twelve-hour forecast here
What-if / optimisation	There is one schedule and nobody is choosing between schedules. Sec. 11.6.3 shows it would *fit* – 3.4 seconds – and fitting is not a reason
Rehearsal	No operator to train, no certification to earn
Command path	Stage 2 is advisory. Chapter 2's sentence stands
3D visualisation	All three of Sec. 11.3.2's tests fail

Five services declined, one and a half built. Chapter 6 Sec. 6.9 chose the cheaper option in five of six subsections and called that the chapter's thesis in miniature; Chapter 8 Sec. 8.7.5 declined two proposals of four. The same thesis, one layer up: **the question is never whether the service would work, it is whether a decision needs it.**

11.10 Faded example: the turbine's services

Chapters 4 through 10 each took the turbine one step further. Now expose it. Two parts are worked; four are yours.

The system, recapped. A floating offshore wind turbine with a reduced-order structural model, a Kalman filter, a fatigue service, Chapter 8's anomaly detector on residuals, Chapter 9's three sampling rates with the connector at the edge, and Chapter 10's three-tier retention. Two outputs at two levels of rigour: maintenance scheduling (advisory) and life extension (certification). Diagnostic twins of this shape run on operational floating turbines [7], and are validated against full-scale prototype measurements [8].

(a) The service set is different from the demonstrator's – worked, because the difference is instructive. Reading Chapter 3's grid for this asset:

Service	Built?	Why
Monitor	Yes	Operations centre watches a fleet
Diagnose	Yes	Chapter 8's anomaly detector, on residuals
Predict	**Yes, and it is the point**	Remaining life is the product
Decide	Advisory only	Vessel scheduling; no closed loop offshore

Service	Built?	Why
Certify-and-train	**Yes, and this is the surprise**	The life-extension output *is* a certification service, and Chapter 1's fifth pattern is where it lives

Four of five, against the demonstrator's one and a half – and the reason is entirely Chapter 1's: the cost of being wrong, and the cost of going there, are both enormous, so more patterns clear the bar.

(b) `cannot_evaluate` matters more here, not less – worked. The demonstrator's connector is a cable to a Pi in the next room. This one is a satellite or cellular link to a structure in the North Sea, and Chapter 9 Sec. 9.10(e) already asked what a hard bandwidth cap does to it. So `cannot_evaluate` will be common, and the consequence is sharper: a maintenance decision deferred because data did not arrive is a vessel not dispatched, and **the cost of not knowing is a number somebody can compute.** Route it, aggregate it, and report its rate as a service-level figure – because at this asset a rising `cannot_evaluate` rate is itself a business event.

Now yours.

(c) Run Sec. 11.4.5's alert economics for this asset. You will need to invent a fault base rate and a false-alarm rate; state them. Then decide whether confirmation (Sec. 11.4.6) is available here, given that a structural anomaly is not a twice-daily event with a natural repeat interval. *Hint: what plays the role of "the next dose"?*

(d) Sec. 11.5.5 said a remaining-life output must carry its range insistently. Specify what the fatigue service returns, for both the advisory output and the certification output, and say what differs between them. *Hint: Chapter 7 Sec. 7.9(a) established these are two credibility arguments, not one.*

(e) Sec. 11.5.4's scoring loop compares a past forecast against what happened. Work out what scoring means for a remaining-life prediction whose horizon is years, and propose something that can be scored on a timescale short enough to be useful. *Hint: what shorter-horizon quantity does the fatigue calculation depend on?*

(f) Sec. 11.6.3 sized the demonstrator's what-if service at 3.4 seconds and concluded no job queue was needed. Do the same arithmetic for a what-if service over this turbine's reduced-order model, using Chapter 6's figures, and say which architectural decision your answer forces.

11.11 Posed problem: the service catalogue

No solution is given.

The situation. You have inherited the district-heating twin of Chapters 9 and 10: 400 substations, three vendors, a fault detector on the supply-return temperature difference. It has been running two years. The data layer has just been rebuilt to Chapter 10's shape, and the operations team is unhappy for reasons nobody has quantified:

- The fault detector raises "roughly forty alerts a week" across the fleet. Nobody knows how many were real, because there is no resolution field.
- Operators say they "mostly ignore" the alerts but cannot say which ones.
- A weekly report of predicted demand is produced by a script on somebody's laptop and emailed as a spreadsheet.
- The utility's board has asked for "an AI dashboard".
- One engineer wants to add automatic valve control "since the model is good now".

Produce a service catalogue and plan of no more than four pages containing:

1. **Each existing and proposed capability mapped to Chapter 1's five value patterns and Chapter 3's component grid**, with what each needs and whether the components exist.
2. **The alert economics** for the fault detector, per Sec. 11.4.5. You will have to state assumptions; state them as assumptions and say which measurement would replace each. Compute the signal ratio and say what it implies about "mostly ignore".
3. **Three outcomes, applied.** How much of the forty-a-week is plausibly `cannot_evaluate`, given Chapter 9's three vendors, and where should it go instead?
4. **A suppression and confirmation strategy** with the detection latency it costs, and an explicit statement of which false alarms it does not remove. *Note that a fault detector on a difference between two sensors has a systematic failure mode – Chapter 9 Sec. 9.11 flagged it.*
5. **What the weekly demand report must gain** to become a forecast service, per Sec. 11.5.1, and whether it should exist at all.
6. **A response to "an AI dashboard"** that is neither a refusal nor compliance: name what the board is actually asking for, in terms of Sec. 11.3.3's four audiences, and say which view answers it.
7. **A position on automatic valve control**, using Sec. 11.6.4 and Chapter 2 Sec. 2.8. State what must exist first, and who owns the safety argument for a municipal heating network.
8. **A build order** with what you would ship in the first month, and one thing you would delete.

What a good answer looks like. It computes the signal ratio before proposing anything, because that number reframes the whole complaint: "we mostly ignore the alerts" is not an attitude problem, it is a correct response to a service with a bad ratio, and the fix is engineering rather than training. It notices that the resolution field's absence means the utility cannot report its own value and puts that first. It says plainly that a fault detector on a *difference* between two sensors cannot be fixed by confirmation alone. It treats "an AI dashboard" as a request for the sponsor's view of Sec. 11.3.3 rather than as a technology request. And its "one thing you would delete" is a real answer – most likely the laptop spreadsheet, which is a service with no owner, no provenance and no scoring.

11.12 Summary

Seven things, tied to the five objectives.

1. **Chapter 3's five patterns are a build order** (Sec. 11.1.1). Monitor is nearly free once the connector and store exist; the jump to diagnose is the largest step and

holds most of the value; only one cell in the whole grid writes to the physical world. *(Objective 1.)*

2. **Four properties separate a twin service from an ordinary one** (Sec. 11.1.2): age awareness, quality awareness, provenance emission, and a defined behaviour when the input is missing. Ask any service what it does when its input is two hours old, marked `substituted`, and absent – three different answers, or it has one property where it needs four. *(Objective 1.)*
3. **A service with two outcomes reports infrastructure failures as physical faults** (Sec. 11.4.2). The third outcome, `cannot_evaluate`, has a different cause, a different owner and a different action – and Chapter 9 built the vocabulary for it two chapters before anything consumed it. *(Objective 2.)*
4. **Chapter 10's bug, fixed** (Sec. 11.4.3). Check that the post-dose operand exists and is after the dose; defer while late data may still arrive; return `cannot_evaluate` when it does not. Re-run against 14 August and the alert does not fire, because the dose had landed. *(Objective 2.)*
5. **One alert in twenty-four would have been real** (Sec. 11.4.5). Twelve genuine episodes, ninety false alarms, and one hundred and seventy-five `cannot_evaluate` – so the largest category is the one a two-outcome service mislabels. Route the third outcome elsewhere, suppress duplicates, and require confirmation, and the queue becomes about thirteen a year with twelve of them real. **Confirmation costs eight to sixteen hours against a ninety-six-hour baseline, and defeats spread but not bias** – which is why Chapter 7's residual monitors are not optional. *(Objective 3.)*
6. **A forecast is not a number** (Sec. 11.5). It carries a horizon, an `as_of`, a range from Chapter 7's hold-out, the scenario it assumed, and its provenance. And the scoring loop – comparing past forecasts against what happened, forever – costs a day and turns Chapter 7's hold-out into a standing measurement. *(Objective 4.)*
7. **Optimisation is a search and it is not learning** (Sec. 11.6.2), its reasoning is inspectable, and Chapter 6's arithmetic sizes it before anybody builds a job queue – 3.4 seconds for five candidate schedules, so no queue. The objective function is a business statement, and an optimiser will enthusiastically minimise whatever it is given. *(Objective 4.)*

And what the demonstrator's service catalogue says about the book's method. One and a half services built, five declined, and the declining took as much reasoning as the building (Sec. 11.9.5). Chapter 6 chose the cheaper option five times in six; Chapter 8 accepted two AI proposals of four and deferred a third; this chapter declines a forecast that would run in 3.4 seconds, *because fitting is not a reason.* **Three chapters of Part III have now reached the same conclusion from three different directions, which is about as much evidence as a book can offer that it is the right one.** *(Objective 5.)*

11.13 Exercises

Solutions or hints follow. Each exercise names the objective it exercises.

11.13.1 *(Objective 1.)* For each, name the value pattern and read off Chapter 3's grid which components it needs: (a) a screen showing every pump's on/off state; (b) a weekly

email of which pots are trending dry; (c) a service that picks the cheapest of five irrigation schedules; (d) a service that tells you a filter will clog in about six days; (e) a training simulator for new lab staff.

11.13.2 *(Objective 1.)* A colleague proposes a "monitor service" that also switches off a pump when moisture exceeds a limit. Give the two-sentence response, using Chapter 3's grid.

11.13.3 *(Objective 2.)* For each situation, say which of the three outcomes the diagnose service should return and who should receive it: (a) the post-dose reading is 604 against an expected step of 56.4 from 606; (b) no post-dose reading exists and the lateness allowance expired an hour ago; (c) the post-dose reading exists but is marked `substituted`; (d) the post-dose reading is 1023, the top of the sensor range; (e) no post-dose reading exists and the dose was two minutes ago.

11.13.4 *(Objective 2.)* Your diagnose service returns two outcomes and you cannot change it this quarter. Name the one change you would make to the *alert message* that recovers most of the value of the third outcome, and say what it still does not fix.

11.13.5 *(Objective 3.)* A fleet of 60 machines produces 4 evaluable events per machine per day. The measured false-alarm rate is 0.5 per cent per event, genuine faults occur 30 times a year across the fleet, and 3 per cent of events cannot be evaluated. Compute the annual totals and the signal ratio. Then apply Sec. 11.4.6's three fixes and recompute, assuming confirmation of two consecutive events.

11.13.6 *(Objective 3.)* Continuing 11.13.5: the events are 6 hours apart, and the requirement is to detect a fault within 24 hours. Is three-event confirmation available? Compute its false-alarm rate and its worst-case latency, and say whether you would use it.

11.13.7 *(Objective 4.)* A forecast service returns `{"moisture_in_12h": 604.2}`. List the four things missing per Sec. 11.5.1, and for each say which decision becomes unsafe without it.

11.13.8 *(Objective 4.)* Classify each as scenario comparison, optimisation, or a learned policy: (a) run the model under five candidate watering schedules and show the operator a ranked table; (b) search 200 schedules and return the one minimising water while keeping moisture above a floor; (c) a network trained on a year of operator decisions that outputs tomorrow's schedule; (d) run the model under the current schedule with the forecast hot week substituted. Then say which two could be swapped for each other with no change to what the operator sees, and why that matters.

11.13.9 *(Objective 5.)* Your sponsor asks for a 3D visualisation of the greenhouse. Apply Sec. 11.3.2's three tests, then write the two-sentence reply – which should neither refuse nor agree, but should name what they are likely to actually want.

11.13.10 *(Objectives 1-5, and the one that leaves the book.)* Take the store you built in exercise 10.13.10 and add a monitor view: the five layers of Sec. 11.3.4, plus age of twin per binding. Then run it against your real week of data and report which of the five layers was hardest to produce, and whether any of them showed you something about your connector that Chapters 9 and 10's checks did not.

Solutions and hints

11.13.1. (a) Monitor – connector, store, thin estimator or none. (b) Monitor, arguably diagnose if "trending dry" is a comparison against a model expectation; the exercise is here because the answer depends on that, and if it is a threshold on a raw value it is monitor and needs no model. (c) Decide – adds the runner at scale; advisory unless it also applies the schedule. (d) Predict – adds the runner. (e) Certify-and-train – models and runner, **and possibly no connector at all**, per Sec. 11.7.1.

11.13.2. "That is not a monitor service; the moment it switches a pump it is the acting form of *decide*, which is the one cell in Chapter 3's grid that needs a command path, an actuation guard, fail-safe behaviour defined per command, and Chapter 7's credibility argument plus a safety argument. If we want that, it is a separate increment with a separate cost – and if we do not, the limit check belongs in the controller, not in the twin."

11.13.3. (a) `fault` – observed step is -2 against an expected 56.4; duty researcher. (b) `cannot_evaluate` – connector owner. (c) `cannot_evaluate` – a substituted value is not evidence about the world (Chapter 9 Sec. 9.6.2); connector owner. (d) `cannot_evaluate`, and additionally an alert on the *sensor*: 1023 is `out_of_range`, which is evidence about the instrument rather than about the pot, and Chapter 4's assumption A2 predicted exactly this at saturation. (e) **Neither – defer.** The lateness allowance has not expired and the window has not closed. This is the case Chapter 10's incident turned on, and a service that returns an answer here is the service Sec. 11.4.3 repaired.

11.13.4. Change the message so that the alert states **the observed step and the readings it was computed from, with their timestamps and qualities**. A human then sees "observed step 0.0, computed from 606 at 17:00 and 606 at 17:00" and recognises an infrastructure failure in two seconds. What it does not fix: the alert still goes to the researcher rather than the connector owner, it still counts against the signal ratio of Sec. 11.4.5, and it still cannot be routed or aggregated automatically – because the distinction exists in prose rather than in a field.

11.13.5. Events per year: `60 x 4 x 365 = 87,600`. Cannot-evaluate: `87,600 x 0.03 = 2,628`. Evaluable events: `87,600 - 2,628 = 84,972`. False alarms: `84,972 x 0.005 = about 425`. Genuine episodes 30, at say 2 raw alerts each = 60. **Total about 3,113 a year, or 60 a week, with a signal ratio of 30 in 3,113 – 1 in 104.** Fix 1 removes 2,628. Fix 2 takes 60 to 30. Queue: `30 + 425 = 455`, ratio 1 in 15. Fix 3: consecutive pairs per machine per year is `1,460 - 1 = 1,459`, times 60 machines is 87,540 pairs; `0.005 x 0.005 = 0.000025`; `87,540 x 0.000025 = about 2.2` confirmed false alarms a year. **Queue: about 32 a year, ratio 30 in 32.** From 1 in 104 to 15 in 16.

11.13.6. Events are 6 hours apart, so two-event confirmation costs 6 hours and three-event costs 12 – both inside a 24-hour requirement, so **yes, it is available.** Three-event false alarms: `0.005^3 = 1.25e-7`, times about 87,480 consecutive triples, is about **0.011 a year** – roughly one per century. Would I use it? Probably not: two-event already gives 2.2 false alarms a year against 30 real faults, so the third event buys about two fewer false alarms a year and costs another 6 hours on every detection. **The marginal return has collapsed, and the right answer is to stop at two** – which is worth noticing, because the temptation with a cheap multiplicative win is to keep going.

11.13.7. Missing: (i) the `as_of` – twelve hours from *when*, which makes the number

unusable for scheduling anything; (ii) the range, so no decision can be made about how much margin to leave, and Chapter 7's entire Sec. 7.6 is discarded; (iii) the assumed scenario, so a user cannot tell whether it assumed today's schedule or an amended one, and any decision that *changes* the schedule invalidates the forecast it was based on; (iv) provenance, so Chapter 7's incident review cannot reconstruct it and Chapter 10's five questions are unanswerable.

11.13.8. (a) Scenario comparison. (b) Optimisation. (c) Learned policy. (d) Scenario comparison – a single what-if, which is the degenerate case. **The two that could be swapped with no change to what the operator sees are (b) and (c)**: both output a schedule. That matters because the operator has no way to tell them apart, while their failure modes are entirely different – (b) fails visibly when the model is wrong and its candidates and scores can be inspected, (c) fails silently off-distribution with nothing to inspect (Chapter 8 Sec. 8.5.5, Sec. 8.6.1). **When two services are indistinguishable at the interface and different in their failure modes, the interface must say which one it is.**

11.13.9. Test 1, is the answer spatial? No – the answer is "pot 3 did not get water". Test 2, is the audience spatial? Marginally: a technician does walk to a pot, but twelve pots on a bench need a label, not a model. Test 3, is the model spatial? No – one number per pot. **All three fail.** The reply: "A 3D view would not answer any question we currently ask, and the plot in Sec. 11.3.4 answers all of them – but I think what you want is a view you can show the funders, which is the value metric of Chapter 1 and its trend, and I can build that this week." *The point of the exercise is that the honest answer is not "no"; it is that a request for a picture is usually a request for a different audience's view.*

11.13.10. No solution. One prediction: the hardest layer will be breaking the line across gaps, because the default in every plotting library is to join the points, and you will discover that your own eye had been believing those lines for a week.

11.14 Where to go next

In this book. Chapter 12 selects the platforms, tools and topology this chapter deliberately named without recommending, and answers the build-versus-buy question for the service layer; twin-as-a-service platforms that generate and host services automatically are one shape of that answer [9], and open-source twin frameworks differ substantially in which of Chapter 3's five services they supply out of the box [10], [11]. Chapter 13 covers the standards, including the service and interface definitions that let a twin's services be discovered rather than documented. **Chapter 14 is where these services are operated**: the work queue of Sec. 11.8.3 when it backs up, the day a threshold changes, and the fault injection and replay that Sec. 11.7.2 named and deferred. Chapter 15 asks what happens when the fleet is 400 substations rather than twelve pots, and Sec. 11.4.5's arithmetic is the part that scales worst – a signal ratio that is tolerable for one asset is an unusable service for four hundred.

In the literature, if you want more.

- *Visualisation:* [1] is a chapter-length treatment aimed at twin builders, covering what visualisation is for, the service forms it takes, and the dimensionality limit

that Sec. 11.3.1 turns into a design question.

- *The decide row, properly:* [6] on what-if analysis, design-space exploration, fault injection and predictive maintenance as twin services – the source for Sec. 11.6.1's definition and the place to go for the machinery this chapter named and did not teach.
- *Prognostics as a service:* [3] for the systematic literature, [4] for a reference architecture, [5] for an overview. These are the Chapter 1 references parked for this chapter.
- *A worked service set on real hardware:* [12] and [13] show the incubator's services – monitoring, state estimation, calibration, decision support – wired together, which is Sec. 11.9 done by somebody else on a different physical twin.
- *Diagnosis in the field:* [7] for the diagnostic service on an operational floating turbine, and [8] for the same asset validated against a prototype.
- *Consulted, not drawn on above:* [2] on how platforms present twins, [14] on the platform properties a service layer should expose, [15] on decision support framings, and [16] on adapting interfaces to the operator – a direction Sec. 11.3.3's four audiences only gestures at.

In the demonstrator. Exercise 11.13.10 is the assignment, and it is the first one that produces something you will look at every day: the five-layer plot, against your own week of data, with the gaps broken. Then, if you want the rest of the chapter, add the diagnose service of Sec. 11.4.3 with its three outcomes and count what it produces over a month. The count is the design review.

11.15 References

[1] 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.

[2] D. Parle, G. Sharma, N. Anand, N. Padgaonkar, D. Stoddart, and D. Malley, "A Comparative Analysis for Harnessing Digital Twin Platforms for Net-Zero Manufacturing," *IEEE Access*, vol. PP, Aug. 2024, doi: 10.1109/ACCESS.2024.3447475.

[3] R. van Dinter, B. Tekinerdogan, and C. Catal, "Predictive maintenance using digital twins: A systematic literature review," *Information and Software Technology*, vol. 151, p. 107008, Nov. 2022, doi: 10.1016/j.infsof.2022.107008.

[4] R. van Dinter, B. Tekinerdogan, and C. Catal, "Reference architecture for digital twin-based predictive maintenance systems," *Computers & Industrial Engineering*, vol. 177, p. 109099, Mar. 2023, doi: 10.1016/j.cie.2023.109099.

[5] D. Zhong, Z. Xia, Y. Zhu, and J. Duan, "Overview of predictive maintenance based on digital twin technology," *Heliyon*, vol. 9, no. 4, p. e14534, Apr. 2023, doi: 10.1016/j.heliyon.2023.e14534.

[6] 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.

[7] F. Stadtmann and A. Rasheed, "Diagnostic Digital Twin for Anomaly Detection in Floating Offshore Wind Energy." arXiv, Jun. 2024. doi: 10.48550/arXiv.2406.02775.

[8] E. Branlard, J. Jonkman, C. Brown, and J. Zhang, "A digital twin solution for floating offshore wind turbines validated using a full-scale prototype," *Wind Energy Science*, vol. 9, no. 1, pp. 1–24, Jan. 2024, doi: 10.5194/wes-9-1-2024.

[9] P. Zech, C. Nardin, S. Ristov, M. Flora, and R. Breu, "Digital-Twins-as-a-Service in Construction Engineering," in *2024 IEEE 20th International Conference on Automation Science and Engineering (CASE)*, Aug. 2024, pp. 3004–3010. doi: 10.1109/CASE59546.2024.10711409.

[10] 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.

[11] 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.

[12] C. Gomes *et al.*, "Digital Twin Tutorial: The Incubator Case Study," in *Engineering Trustworthy Software Systems: 6th International School, SETSS 2024, Chongqing, China, April 14–21, 2024, Tutorial Lectures*, J. P. Bowen, C. Gomes, and Z. Liu, Eds., Singapore: Springer Nature, 2025, pp. 68–101. doi: 10.1007/978-981-96-4656-2_3.

[13] 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.

[14] 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.

[15] J. Li, J. Grübel, A. Nadi, M. Snelder, B. van Arem, and J. Gao, "Digital Twin Federation for Urban Mobility Assessment: A Functional Architecture for Low-Car Transformations in the Netherlands," 2025, Accessed: Oct. 25, 2025. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5370630

[16] F. Franco, L. Lamazzi, M. Picone, M. Savarese, C. A. Grazia, and L. Bedogni, "Customizing Human Machine Interfaces leveraging Digital Twins and Large Language Models," in *2026 IEEE 23rd Consumer Communications & Networking Conference (CCNC)*, Jan. 2026, pp. 1–6. doi: 10.1109/CCNC65079.2026.11366401.