Chapter 12 – Platforms and Composability: Buying, Building, and Assembling from Parts

12.0 Before you start

Where we are. Chapters 9, 10 and 11 built the connector, the store and the services – and each of them, at a decisive moment, refused to name a product and pointed here. Chapter 9 declined to compare protocols and said "Chapter 12 selects tools". Chapter 10 named no database engine. Chapter 11 named no framework. Chapter 6 sent tool and platform selection here before Part III had even started.

That is four chapters of deferral pointing at one question, and this chapter has to answer it without doing the thing that would make it useless.

The problem, stated plainly. A textbook that names products is wrong in two years. A textbook that names none breaks four explicit promises. Both failures are real, and the second is not the safer one it looks like.

The resolution is a pattern this book already used. Chapter 9 Sec. 9.4.3 faced the same problem with protocols and solved it by evaluating transports against *what a twin's downstream components break without*, rather than against protocol features – a table that survives any protocol list because it is keyed on the twin's needs. This chapter does the same thing one level up:

- **What does the platform supply?** Counted in Chapter 3's seven components, at three levels (Sec. 12.2).
- **What does it take away?** Seven properties, one from each of Chapters 7 to 11, each naming what breaks in its absence (Sec. 12.3).
- **What are the things called?** Categories, one line each, explicitly dated, no comparison (Sec. 12.4).
- **Should you buy at all?** A cost model with five named parameters, seeded from Chapter 1's own build estimate (Sec. 12.5).

  This chapter names no product and compares none. It supplies a method for evaluating whatever exists when you read it.

You are not alone in finding this hard. The field says so about itself: work on evaluating twin platforms reports that previous studies have paid little attention to evaluation methods at all, and that the few that exist are "mostly based on subjective opinions, simple decision-making methods, and general-purpose criteria" [1]. That is the same admission Chapter 9 quoted about publish/subscribe systems, and it licenses supplying a method rather than a survey.

What you are assumed to know. Everything so far. Especially: Chapter 3's seven components, its five services, the platform view of Sec. 3.5.4 and the three deployment forces of Sec. 3.6; Chapter 1's cost and payback estimate in Sec. 1.8.7; Chapter 7's credibility argument and its contract clauses; Chapter 8's three model artifacts; Chapter 9's five transport properties and six quality states; Chapter 10's context history, provenance chain and dataset definition; Chapter 11's three-outcome service and its declined service list.

The maths budget. As Chapters 9 to 11: more arithmetic welcome, no new mathematics.

One set piece, the cost model of Sec. 12.5, and it is multiplication throughout.

What this chapter deliberately does not cover. Any named product. Standards and the asset-model specifications – Chapter 13. Ecosystems, data spaces and twin-of-twins – Chapter 15. Operational running, migration and upgrade – Chapter 14. Container, orchestration and serverless technology – the topology *decision* is here, the technology is not. Procurement and contract law – Chapter 7 Sec. 7.7.5 already specified the credibility clauses. Security architecture – Chapter 3 Sec. 3.3.3 and Chapter 13.

Learning objectives

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

1. **Determine** which of Chapter 3's seven components a candidate platform supplies, and at which of three levels.
2. **Evaluate** a platform against seven properties earlier chapters established, and **identify** which of them a platform can take away without anybody noticing until it matters.
3. **Compute** build-versus-buy from a five-parameter cost model, and **determine** which two of its parameters decide whether a platform ever pays off, as distinct from how soon.
4. **Explain** what composability means beyond reuse, and **name** the four seams that make a part replaceable.
5. **Decide** deployment placement per component, and **state** what changes when a platform makes that decision for you.

12.1 The question four chapters deferred

12.1.1 What was actually promised

Worth listing, because the pattern in the list is itself informative:

Promised in	The words
Ch3 Sec. 3.5.4	the platform view is "our anatomy plus the question which of these do you build, and which do you consume – which is Chapter 12's question"
Ch6 Sec. 6.14	"the platforms and tools this chapter named without recommending"
Ch7 Sec. 7.13	"Chapter 12 asks what a platform gives you of all this"
Ch9 Sec. 9.4.4	"Chapter 12 selects. Chapter 13 covers the standards."
Ch10 Sec. 10.14	"Chapter 12 selects the engines this chapter deliberately declined to name"
Ch11 Sec. 11.14	"Chapter 12 selects the platforms, tools and topology this chapter deliberately named without recommending"

Six chapters, one verb: select. And notice what each of them was protecting when it deferred. Chapter 9 wanted to teach what a twin needs from a transport rather than what the Message Queuing Telemetry Transport (MQTT) does. Chapter 10 wanted to teach what a store must represent rather than which engine represents it. Chapter 11 wanted to teach what a service must answer rather than how to expose it.

Each of those chapters produced a durable list by refusing to produce a product list, and this chapter's job is to make those lists into a purchasing decision rather than to replace them with a shopping catalogue.

12.1.2 Why a product table would be worse than nothing

Three reasons, and the third is the one that matters.

It dates. Obvious, and the least serious, because a reader can tell that a 2026 table is a 2026 table.

It flatters the wrong axis. Product tables compare features, and features are what vendors choose to have. The properties that decide whether a twin survives its first incident review – Sec. 12.3's seven – appear on no vendor's feature list, because they are not features. They are consequences of design decisions that a datasheet does not record.

It substitutes for the evaluation. A reader with a table picks from the table. A reader with a method evaluates the three things actually available to them, in their industry, at their budget, against their twin. **The second reader is doing engineering and the first is doing shopping**, and only one of those transfers to the situation you will actually be in.

12.1.3 What is genuinely known

Two useful things, and it is worth being clear that they are the extent of it.

The same components recur under different names. Surveys of open-source twin frameworks find Chapter 3's anatomy appearing again and again with different vocabulary [2], which is what makes a component-based evaluation possible at all. If every platform were shaped differently, Sec. 12.2's checklist would not work.

Fragmentation is itself the obstacle. Work on twins across the edge-to-cloud continuum argues that the fragmentation of protocols, formats and deployment architectures is the thing standing in the way, rather than any missing capability [3]. That is worth knowing before a meeting in which somebody proposes solving fragmentation by adopting one more platform.

12.2 What a platform supplies, counted properly

12.2.1 Chapter 3's seven components are the unit of account

Not features, not modules, not "capabilities". Chapter 3 gave you seven components and Part III has now built four of them in detail, so you know what each one costs to build and what it must do. That makes them the right currency.

Component	Built in	Typical platform coverage
Connector	Ch9	Often good for common protocols, poor for yours
Store	Ch10	Almost always supplied, and Sec. 12.3 is about what it supplies *badly*
State estimator	Ch3, Ch6	Rarely; usually your code, hosted
Models and registry	Ch3, Ch4-8	Registry sometimes; models never
Simulation runner	Ch5, Ch6	Sometimes, and often the reason to buy
Services	Ch11	Monitor and visualisation almost always; diagnose rarely; the rest never
Command path and guard	Ch3	Almost never, and **you should be suspicious when it is**

Read the right-hand column as the chapter's first finding. Platforms cluster around the components that are the same for everybody – store, runner, monitor, visualisation – and thin out at exactly the components that carry your twin's value. Chapter 3 Sec. 3.4 said the jump from monitor to diagnose is the biggest step in the grid and where most of the engineering value per euro sits. **That step is the one platforms help least with**, and it is not a criticism of platforms: a diagnose service encodes your value metric, your model and your thresholds, and nobody can supply that.

12.2.2 Three levels of "supplies"

"Does it support X" is unanswerable. Three levels are answerable:

Hosts. It runs your code for that component. You still write it; the platform gives it somewhere to live, a deployment story and, usually, operations.

Implements. It provides the component. You configure it rather than writing it.

Opinionates. It provides the component *and* constrains how the neighbouring components may work – its own data model, its own event shape, its own service contract.

Figure 12.1 draws the three, and what the third one does to its neighbours is the reason the distinction matters commercially.

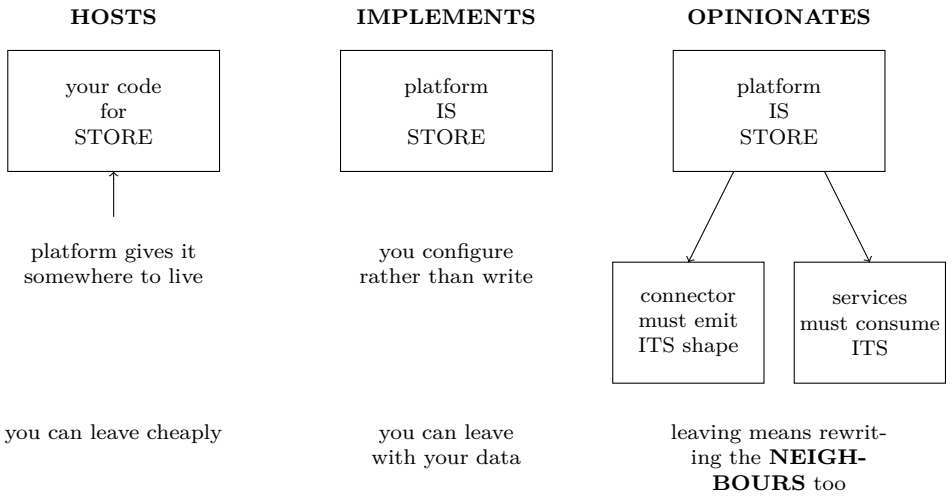

Figure 12.1 Three levels, and only the third one reaches sideways. "Does it support X" cannot distinguish them, which is why the question is unanswerable.

The third level is the one to look for, and the one nobody advertises. A platform that *implements* a store is a convenience. A platform that *opinionates* a store has decided your schema, and Chapter 10 spent a whole chapter on four things a twin's schema must do that an ordinary one need not. If the platform's store cannot represent Chapter 9's six quality states or Chapter 10's validity intervals, you have not bought a store – you have bought a constraint on every downstream chapter.

How to find out in an hour. Take Chapter 10 Sec. 10.9.2's eleven tables and try to express them in the platform's data model. The tables that will not fit tell you exactly which level you are at, and which properties of Sec. 12.3 you are about to lose.

12.2.3 What "assemble" means as a third option

The title names three options and most discussions have two. The third is real and it is the one this book's own worked example takes:

Option	What you do	What you are betting on
Build	Write all seven components	That your twin is unusual enough that generic parts do not fit
Buy	Take a platform, configure it, write the rest	That the platform's opinions match yours, and that its fixed cost amortises
Assemble	Take ordinary components – a time-series store, a broker, a plotting library, a scheduler – and wire them yourself	That the seams are yours, so you can replace any part

Assemble is not a compromise between the other two, and treating it as one is the common mistake. It is a different bet: that the *integration* is the cheap part and the

lock-in is the expensive part. Sec. 12.5 shows when that bet pays, and Sec. 12.6 shows what makes it work when it does.

Open-source twin frameworks exist precisely to make this option available to organisations that cannot afford a commercial solution [2], and the platform-as-a-service proposals in the literature are explicitly attempts to make assembly reusable – building twins from reusable components and offering them as a service, rather than starting from scratch each time [4].

12.3 Evaluate a platform by what your twin breaks without

This is the chapter's core, and it is Chapter 9 Sec. 9.4.3's table one level up. **Seven properties, one drawn from each of Chapters 7 to 11.** Each row names what breaks in its absence, and none of them appears on a feature list.

12.3.1 The seven properties

#	Property	The question to ask	What breaks without it
1	**Event time preserved end to end**	Does a reading keep the time the *world* was that way, from ingest to query, or does something overwrite it with arrival time?	Every comparison in Chapter 7. Chapter 5's replay. Chapter 6's estimator
2	**Absence representable**	Can I store "no reading, and here is why" as distinct from a null and from a zero? Are there more than two states?	Chapter 11's third outcome. Chapter 7's calibration exclusions. Chapter 9's whole Sec. 9.6
3	**Context with history**	Can I ask which sensor was bound to this stream *on 14 August*, or only which one is bound now?	Any interpretation of past data. Chapter 10 Sec. 10.4.2

#	Property	The question to ask	What breaks without it
4	**Provenance out, not just in**	Given an output, can I get the model version, parameter set, input watermark and decision time that produced it – **through an interface, not through a support ticket**?	Chapter 7's five incident questions. Chapter 8's retroactive invalidation
5	**Datasets nameable and reproducible**	Can I define a dataset and get the same rows back next month, including a frozen ingest watermark?	Chapter 7's hold-out. Chapter 8's retraining. Chapter 10 Sec. 10.7
6	**Model versions as first-class**	Can a version identify code, weights and training data, and can two versions run side by side against the same scenario?	Chapter 8 Sec. 8.2.3. Chapter 7's golden trajectories
7	**An enforcement point at the boundary**	Is there somewhere my code runs *before* the model is called, so I can refuse an out-of-envelope input?	Chapter 7's validity envelope. Chapter 8's input-coverage check

How to use this table. Not as a scorecard with weights – that is the "simple decision-making method" the field is already criticising itself for [1]. Use it as seven questions with concrete answers, and record the answer, not a score. "Property 3: no – context is a mutable record, history requires an external audit log we would build" is a usable finding. "Property 3: 2 out of 5" is not.

And notice the asymmetry that makes this table worth the page. A missing component you can build. A missing *property* you often cannot, because it is a consequence of the platform's internal design and no amount of your code puts event time back once something has overwritten it.

12.3.2 The exit question, which decides lock-in

One more question, and if you only ask one, ask this:

> **Can I get my data out – measurements, context with its history, derived output with its provenance – in a form another system can consume, without the vendor's help?**

Three reasons it is the best single question.

It is testable in an afternoon. Export a month, load it somewhere else, and see what survived. Do this during the evaluation, not during the migration.

It subsumes several of the seven. Data that exports without its event time, its quality states, its context history or its provenance has told you about properties 1 to 5 in one experiment.

It is the only question whose answer you need on the worst day. Platform decisions are reversed – because a vendor is acquired, a price changes, a project moves. **The cost of that reversal is set entirely by this question**, and it is fixed on the day you choose, not on the day you leave.

Interoperability at this boundary is exactly what the standards work is for, and Chapter 13 covers it [5], [6]. What belongs here is the engineering habit: run the export before you sign.

12.3.3 What the evaluation looks like in a day

Neither a spreadsheet nor a pilot. One day, four activities:

1. **The component checklist** (Sec. 12.2.1), at the three levels. One hour.
2. **The schema fit**: try to express Chapter 10 Sec. 10.9.2's tables in the platform's data model. Two hours, and it usually settles properties 1, 2 and 3.
3. **The seven questions** (Sec. 12.3.1), answered in prose. Two hours.
4. **The export test** (Sec. 12.3.2). Two hours, and it is the one people skip.

And the output is a page, not a recommendation: what it supplies, at what level, which properties it costs you, and what leaving would take. The decision follows from that page plus Sec. 12.5's arithmetic, and it belongs to whoever owns Chapter 1's value metric.

12.4 The categories, for recognition only

Chapter 9 Sec. 9.4.4 listed protocols this way and the same rules apply: this is for placing a name in a meeting, it is explicitly a snapshot, and it contains no comparison.

Category	What it is	Where it fits Chapter 3's grid
Industrial IoT / Supervisory Control and Data Acquisition (SCADA) platforms	Grown up from plant-floor data collection	Strong connector and store; monitor and visualisation; little else

Category	What it is	Where it fits Chapter 3's grid
Cloud vendors' twin offerings	A graph of entities plus data plumbing on a hyperscaler	Store, context, services scaffolding; models are yours
Simulation vendors' twin products	Grown down from a simulation tool	Strong runner and models; connector and store often thin
Twin-specific open-source frameworks	Purpose-built, community-scale	Varies widely; surveys show the same components under different names [2]
Twin-as-a-service platforms	Twins assembled from reusable parts and offered on demand	The platform view of Chapter 3 Sec. 3.5.4, made concrete [4], [7]
General components, assembled	A time-series store, a broker, a scheduler, a plotting library	Whatever you wire; the seams are yours (Sec. 12.6)

Three observations that will outlast the categories.

Every category is strong where it came from. A platform's origin predicts its shape better than its marketing does, and asking "what was this before it was a twin platform?" is worth more than reading its feature list.

Nobody is strong at the estimator, the diagnose service or the command path. Those encode your physics, your value metric and your risk, and Sec. 12.2.1's right-hand column already said so.

Open-source and commercial is not the axis you think. The axis that matters is Sec. 12.2.2's three levels. An opinionated open-source platform constrains you as thoroughly as a commercial one; the difference is that you can read why, and fork it.

12.5 Build, buy, assemble: the arithmetic

12.5.1 A cost model with five parameters

Every build-versus-buy discussion is really about five numbers, and naming them turns an argument into a calculation. All five are estimates and all five are yours to replace.

```
B  = engineer-days to build one twin from scratch
F  = fixed cost of the platform: learning it, mapping your model onto its
     abstractions, and fighting the parts that do not fit. Paid once
r  = fraction of B the platform removes
a  = cost of your Nth assembled twin as a fraction of B (internal reuse)
p  = cost of the Nth platform-hosted twin as a fraction of B (configuration)

Assemble(N) = B x (1 + a(N-1))
Platform(N) = F + B(1-r) + B x p x (N-1)
```

Seed them from something real rather than from feeling. Chapter 1 Sec. 1.8.7 estimated the demonstrator's build at **six engineer-weeks**, line by line – data path 1.5 weeks, model and fitting 1.5, residuals and alerting 1.5, deployment and handover 1.5. That is **B = 30 engineer-days**, and it is the book's own number rather than an invented one.

For the other four, use these as starting values and argue with them:

```
F = 15 engineer-days     three weeks to learn a platform and map onto it
r = 0.4                  it removes 40 per cent of the build
a = 0.3                  your second twin costs 30 per cent of your first
p = 0.15                 a configured twin costs 15 per cent of a built one
```

12.5.2 The demonstrator, computed

```
Assemble(N) = 30 + 9(N-1)
Platform(N) = 15 + 18 + 4.5(N-1) = 33 + 4.5(N-1)
```

Twins	Assemble	Platform	Cheaper
1	30	33	Assemble, by 3 days
2	39	37.5	Platform
3	48	42	Platform
5	66	51	Platform
10	111	73.5	Platform
20	201	118.5	Platform

The crossover is between the first twin and the second. For one twin of this size, assembling is cheaper – and by three days out of thirty, which is well inside the error of any estimate here. **The honest reading is that at one twin it is too close to call on cost, and at two twins the platform is already ahead.**

That result deserves a sentence, because it explains a pattern you will recognise: platform vendors sell to organisations with fleets, one-off projects hand-build, and both are behaving rationally.

12.5.3 The sensitivity that actually decides it

Vary one parameter at a time, and something structural appears.

Parameter changed	Crossover moves to
F = 30 instead of 15 (a harder platform)	5 twins
r = 0.6 instead of 0.4 (it removes more)	1 twin – ahead immediately
p = 0.3 instead of 0.15 (configuration is fiddly)	**never**
a = 0.15 instead of 0.3 (you are genuinely good at reuse)	**never**
a = 0.5 (each twin nearly from scratch)	2 twins, as in the base case

Two rows say "never", and they say it for the same reason. Look at the slopes rather than the totals. Assembling grows at `a x B` per extra twin; the platform grows at `p x B`. So:

```
Assemble grows by  a x B  per twin  = 9.0 days at a = 0.3
Platform grows by  p x B  per twin  = 4.5 days at p = 0.15
```

Figure 12.3 is why "never" is a possible answer at all: the two options are straight lines, so if the slopes point the wrong way they never meet.

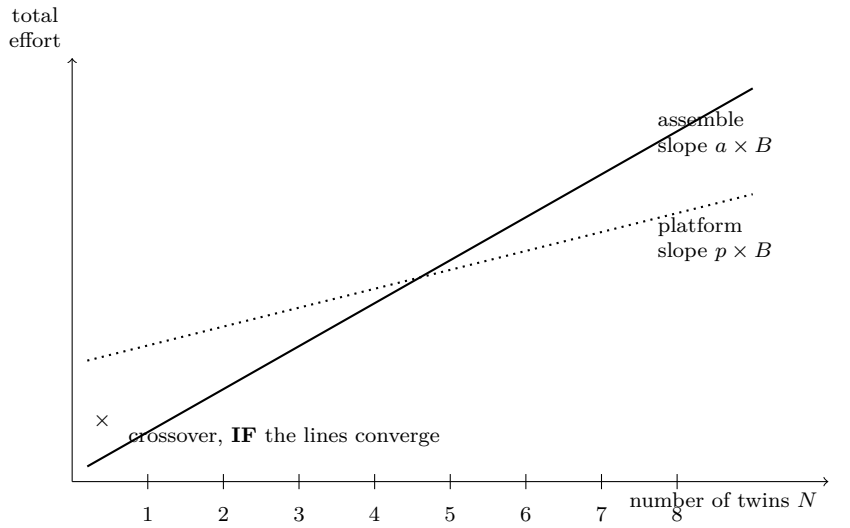

Figure 12.3 Build/buy as two straight lines. The sensitivity table's two "never" rows are not extreme cases; they are the lines diverging.

Two quantities decide it, and they do different jobs.

`rB - F` is the platform's **head start** at the first twin. Positive and it is already ahead before any fleet exists.

`a - p` is the rate its lead **grows** per extra twin. Positive and it pulls away with fleet size; negative and it falls behind.

Written out, the platform is cheaper at `N` exactly when

```
(F - rB) + (p - a) x B x (N-1)  <  0
```

So the two failure modes are different and both are real. A platform with no head start needs `p < a` to ever pay off, and if `p` is not below `a` it never catches up at any fleet size, however good its other numbers. But a platform with a large enough head start – `rB` comfortably above `F` – can stay cheaper across any fleet you will actually build even when `p > a`, because you run out of twins before the slope catches it. Exercise 12.12.5 is that case.

What is worth knowing before a negotiation is that `F` and `r` set the head start and `a` and `p` set the slope, and a vendor will argue about `F` and `r`. Those are the two you can

verify cheaply and the two that stop mattering as the fleet grows.

And that comparison has one side on each side of the table. `p` is a fact about the platform, and it is measurable: configure a second twin during the evaluation and time it. `a` is a fact about your team, and it is also measurable – Sec. 12.6.3 says how, and exercise 12.12.6 makes it concrete.

> **So the build-versus-buy question is, at scale, a question about whether your reuse is better than the platform's configuration.** If you will genuinely build reusable parts – with real seams, so the second twin reuses them without a fork – assembling can win at any scale. If your second twin will be a copy-paste of the first with edits, `a` is nearer 0.5, `p < a` comfortably, and the platform wins almost immediately.

And the uncomfortable part: most teams overestimate `a`. Everybody intends to build reusable parts. The measured version of that intention is what your *previous* project's second instance actually cost, and that number is usually available and usually higher than anybody remembers. Sec. 12.6 is about what would make it lower.

12.5.4 What the arithmetic does not capture

Four things, and skipping them is how a correct calculation produces a bad decision.

Operations. The model counts build effort. A platform usually also runs the thing, and Chapter 14 is about what running it costs. For a small team without an on-call rotation, that can dominate everything above.

The exit cost. Sec. 12.3.2's question has a price, and it is not in the model because it is not paid until it is paid. Add it as a risk-weighted line rather than pretending it is zero.

The properties you lose. Sec. 12.3's seven are not costs in days. A platform that cannot answer Chapter 7's five incident questions has not made your project cheaper; it has made a different, worse project cheaper.

The value case, which is the actual constraint. Chapter 1's payback was nine months on a build cost of EUR 12,000 – that is, on the 30 engineer-days above. **A platform that adds 3 engineer-days to a one-twin project moves the payback by about a month; one that adds 30 doubles the build cost and pushes payback past 18 months, which Chapter 1 Sec. 1.8.8 already identified as the point where the honest recommendation is a cheaper alarm instead of a twin.** Run the platform's cost through Chapter 1's canvas before running it through anything else.

12.6 Composability, which is the third option done properly

Sec. 12.5.3 found that the whole decision turns on `a`, the reuse fraction. This section is about what makes `a` small, which is the only lever in the model you control directly.

12.6.1 What composable means beyond "reusable"

Reuse is using the same part twice. Composability is stronger: **parts that can be combined in arrangements nobody planned, and replaced without touching**

their neighbours.

The distinction matters because the first is achievable by copying and the second is not. A composable approach is argued for in twin practice explicitly as the route to scale – effort gets re-used, results arrive sooner, the approach generalises, and its dynamic range covers both simple and complex cases across an enterprise [8]. The platform proposals make the same bet from the supply side: a generic platform from which twins are built out of reusable components, rather than marshalling assets, models, data and services from scratch each time [4].

The test. Take your second twin. If it needed a change *inside* a part you intended to reuse, that part was reusable and not composable, and your `a` is higher than you think.

12.6.2 The four seams that matter for twins

A seam is a boundary at which a part can be replaced without touching its neighbours. Twins have four that pay, and each is something an earlier chapter already made you build. Figure 12.2 places all four on Chapter 3's anatomy, so the seams are visible as *cuts* rather than as a list.

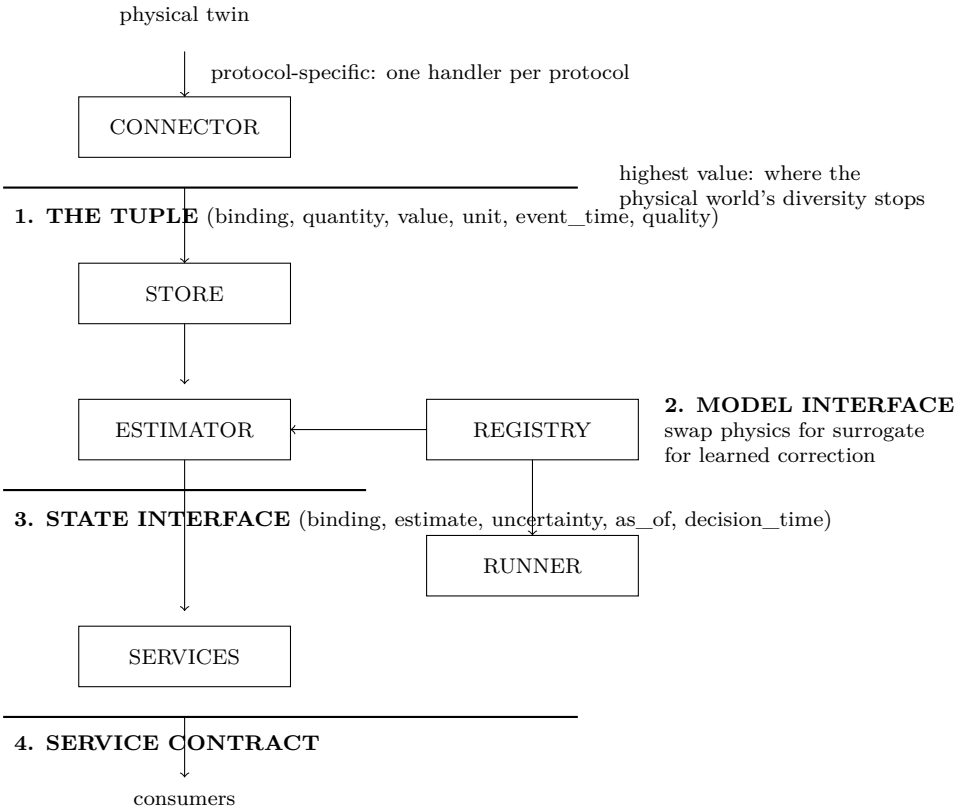

Figure 12.2 Four seams, drawn as cuts across the anatomy. Each one is a place a part can be replaced; together they are what makes "assemble" a bet rather than a hope.

The figure is also the exit plan of Sec. 12.3.2 in advance: a platform that opinionates across one of these cuts has removed a seam you were counting on, and the cost of leaving is measured in how many of the four survive.

1. The tuple, between connector and store. Chapter 9's `(binding, quantity, value, unit, event_time, quality)`. Everything upstream of it is protocol-specific;

everything downstream is not. **This is the highest-value seam in the whole system** because it is where the physical world's diversity stops, and Chapter 9 Sec. 9.4.5 already required one handler per protocol behind it.

2. The model interface, between registry and runner. Chapter 3's model registry plus Chapter 6's Functional Mock-up Interface (FMI) discussion. A runner that can call any model satisfying one interface lets you swap a physics model for a surrogate for a learned correction without touching anything else – which is exactly what Chapter 8 Sec. 8.6.2's physics-only fallback needs at runtime.

3. The state interface, between estimator and services. Chapter 3 Sec. 3.7.3's `(binding, estimate, uncertainty, as_of, last_measurement_at)`, now with Chapter 10's `decision_time`. Services that consume this tuple do not care how the estimate was produced.

4. The context boundary, between the twin and its configuration. Chapter 3 Sec. 3.3.1's rule that binding is data, not code, plus Chapter 10 Sec. 10.4.3's validity intervals. **This is the seam that decides whether your second twin is a deployment or a fork.** If pot 3 versus pot 7 is configuration, twin number two is a row. If it is code, twin number two is a project.

> **Notice that all four seams were specified by earlier chapters for entirely different reasons** – protocol isolation, model swapping, service decoupling, audit. Composability is not an additional discipline layered on top; it is what you get when the boundaries each chapter needed for its own reasons happen to line up. **That is also why retrofitting it is so expensive: the seams are load-bearing everywhere at once.**

12.6.3 Why internal reuse usually fails

Three causes, in the order they occur.

The first twin is built before the second is imagined. Nobody designs a seam for a case they have not met. The mitigation is not foresight; it is **refactoring at the second twin rather than copying** – which costs more than copying, once, and is the entire difference between `a` = 0.15 and `a` = 0.5.

The parts are reusable and the glue is not. Teams extract libraries and then write bespoke wiring for each twin, and the wiring is where the effort was. The four seams above are seams in the *glue*, not in the parts, which is why they are the ones that matter.

Nobody owns the shared parts. A library used by three twins and owned by none accumulates three sets of special cases and becomes a fork by attrition. This is an organisational failure and it is the commonest one – which is why Sec. 12.5.3's `a` is a question about your team.

And the honest note. If none of the three mitigations is available to you – no time to refactor, no owner for shared parts, no second twin in sight – then `a` is high, Sec. 12.5.3's fourth row does not apply to you, and buying is the right answer. **Knowing that about your own organisation is worth more than any platform comparison.**

12.7 Deployment topology, when you are buying rather than architecting

12.7.1 Chapter 3's three forces are unchanged

Chapter 3 Sec. 3.6 already settled the architecture question: latency to the decision, availability of the link, and cost and scale of simulation – and the rule that **deployment is per component, not per twin**, because at least one component always wants to be somewhere else. Nothing about buying changes any of that.

The recurring answer stands too: connector and buffering at the edge, store and heavy compute centrally, services wherever the users are. Work on twins across the edge-to-cloud continuum treats the placement of each part as the design variable rather than as a binary choice [3], [9].

12.7.2 What changes when the platform decides

One thing, and it is worth naming because it is the commonest surprise: **a platform can take the per-component decision away from you.**

If the platform's connector must run in its cloud, then Chapter 9 Sec. 9.8.2's store-and-forward buffering is no longer yours to place – and the demonstrator's happy position, where the Pi keeps a local database and the API takes `since_timestamp`, becomes a happy accident you are relying on rather than a design you own. If the platform's runner is central, Chapter 5's real-time factor for an edge decision is bounded by a round trip that Chapter 3 Sec. 3.6's first force said should not be in the loop.

So add one question to Sec. 12.3.1's seven, for any twin whose components do not all want to live in the same place:

> **Which components can I place, and which does the platform place for me?**

A platform that places all seven for you is a hosting decision disguised as an architecture decision, and Chapter 3's grid is how you notice.

12.7.3 Edge, cloud and federated: three questions, not three answers

Chapter 8 Sec. 8.3.7 named these and handed them here. Each reduces to one question.

Edge. *Does a decision have to happen faster than a round trip, or survive the link being down?* If neither, the edge is a cost with no benefit. The demonstrator's answer is no on both counts; the turbine's is yes on both, and Chapter 9 Sec. 9.8.1's 2.3 TB per turbine per year settles it independently. Work pairing constrained devices with twins that offload computation and validate inferences is one shape of the answer [10].

Cloud. *Is the compute elastic demand real?* Chapter 6 sized the demonstrator's what-if at 3.4 seconds (Chapter 11 Sec. 11.6.3), so no. A fleet running Monte Carlo overnight is a different answer.

Federated. *Is there data you cannot move?* Commercially sensitive, legally restricted, or merely too large. Federated approaches let multiple parties contribute to a model without pooling raw data, with communication-efficient variants for twin settings [11]. **If the**

answer is no, this is complexity with no purchaser, and it is the one of the three most often adopted for reasons other than the question.

12.8 Worked example: the demonstrator's platform decision

12.8.1 What we are deciding

By the end of Chapter 11 the demonstrator needs: a connector polling one HTTP endpoint with five bindings; two stores holding eleven tables and under a gigabyte for a decade; one diagnose service; one five-layer plot; three expiry monitors; and a three-parameter model. Chapter 1 costed the build at 30 engineer-days.

12.8.2 The component checklist

Running Sec. 12.2.1 against any of Sec. 12.4's categories gives roughly the same answer for a twin this shape:

Component	Would a platform help?
Connector	**No.** The source is one HTTP endpoint with a history parameter. Chapter 9 Sec. 9.9.1 scored it five out of five on the properties that matter. A platform's connector is built for protocols we do not have
Store	**Marginally.** 139 MB a year. Chapter 9 Sec. 9.8.1 said explicitly that this volume forces no architecture decision, and Chapter 10 designed eleven tables that any ordinary engine holds
State estimator	No – ours, either way
Models and registry	Registry: marginally. Models: ours
Simulation runner	**No.** A run is 32 microseconds
Services	Monitor plot: yes, a little. Diagnose: no – it encodes our value metric, our threshold and Chapter 11's three outcomes
Command path	Not applicable; Stage 2 is advisory

Nothing in that column says "yes, substantially".

12.8.3 The arithmetic

Sec. 12.5.2, with the demonstrator's own numbers: one twin, `B` = 30, so assemble costs 30 days and a platform costs 33. **Too close to call on cost alone**, exactly as Sec. 12.5.2 predicted.

So the decision falls to the other three considerations of Sec. 12.5.4, and they are not close:

Properties. Chapter 10 Sec. 10.3.3's quality column and Sec. 10.4.3's validity intervals are unusual requirements. A platform that opinionates its store is likely to cost us property 2 or property 3, and Chapter 11's whole three-outcome design depends on property 2.

Value case. Chapter 1's payback is nine months at EUR 12,000. Adding a licence and 15 days of integration to a 30-day build is not a rounding error; it is most of a second payback period.

Second twin. There is no second twin. If the greenhouse expands to a second bench next year, Sec. 12.5.2 says revisit – and Sec. 12.6's seams are what make that revisit cheap.

12.8.4 The decision, and its expiry

> **Assemble.** Ordinary components: a time-series store, an ordinary relational database, a scheduler, a plotting library, and about 30 engineer-days of our own code across Chapters 9 to 11. Build the four seams of Sec. 12.6.2 deliberately, even though there is one twin, because they cost little now and they are what makes `a` small later.
>
> **Revisit when any of these becomes true:** a second twin is funded; the data volume rises by two orders of magnitude; somebody wants the certify-and-train services Chapter 11 declined; or the team loses the person who owns the shared parts.

That last trigger is not a joke. Sec. 12.6.3's third failure is organisational, and a build decision that depended on internal reuse is invalidated by the departure of whoever was doing the reusing. It belongs in the decision record next to the technical triggers.

12.8.5 What this looks like written down

One page, per Sec. 12.3.3: the component checklist, the seven properties answered in prose, the export test result, the arithmetic with its five parameters stated, and the expiry triggers. **Filed where Chapter 14 will find it**, because the day one of those triggers fires is the day somebody needs to know what was decided and on what basis – which is Chapter 10's provenance argument applied to a decision rather than to a number.

12.9 Faded example: the turbine fleet

Chapters 4 through 11 each took the turbine one step further. Now decide how to build it. Two parts are worked; four are yours.

The system, recapped. Eighty floating offshore wind turbines. Per turbine: a reduced-order structural model, a Kalman filter, a fatigue service, an anomaly detector on residuals, three sampling rates with feature extraction at the edge, three-tier retention, and four of Chapter 3's five services – including a certification output with a regulator on the other side.

(a) The arithmetic inverts – worked. Estimate `B` for one turbine twin at 200 engineer-days. **That figure is invented, unlike the demonstrator's 30, which came from Chapter 1 Sec. 1.8.7 line by line** – and what carries the argument is its *ratio* to 30

rather than its absolute value: this twin is roughly seven times the build, because the edge connector alone is harder than the demonstrator's entire build and Chapter 11 Sec. 11.10 found four services where the pot had one and a half. Substitute your own `B` and the shape of the conclusion does not move. With the same `F` = 15, `r` = 0.4, `a` = 0.3, `p` = 0.15:

```
Assemble(80) = 200 x (1 + 0.3 x 79) = 200 x 24.7 = about 4,940 engineer-days
Platform(80) = 15 + 120 + 200 x 0.15 x 79 = 15 + 120 + 2,370 = about 2,505
```

A factor of two, and the ratio is the point rather than the absolute figures: this is a multi-year programme either way, and the difference is of the order of ten engineer-years. At this scale the platform question stops being a build-versus-buy decision and becomes an organisational one, which is what `van_schalkwyk_achieving_2023` means by composability as the route to scale [8] and what work on industrialising twin production is responding to [12].

(b) But property 4 may veto it – worked, because it is the answer people miss. Chapter 7 Sec. 7.9 established that the life-extension output feeds a certification decision with a regulator. Sec. 12.3.1's property 4 – provenance out, through an interface – is therefore not a convenience here; it is a regulatory requirement, and Chapter 8 Sec. 8.6.3's retroactive-invalidation traversal has to run across eighty assets and fifteen years. **A platform that scores well on components and fails property 4 is not cheaper, it is unusable**, and Sec. 12.5.4 said exactly this in the general case. The arithmetic in (a) is necessary and not sufficient.

Now yours.

(c) Run Sec. 12.2.2's three levels against the "simulation vendors' twin products" category for this asset. Where would an opinionated runner help, and where would an opinionated *store* hurt, given Chapter 10 Sec. 10.10's three-tier retention?

(d) Sec. 12.6.2 named four seams. Which is the most valuable for a fleet of eighty nominally identical assets, and what does Chapter 10's context model have to look like for the eighty-first turbine to be a row rather than a project? *Hint: nominally identical is not identical – mooring configurations differ.*

(e) Apply Sec. 12.7.2's added question. Which of the seven components must be placed at the turbine, which centrally, and what happens to your platform shortlist once you require that split?

(f) Sec. 12.8.4 wrote expiry triggers for a decision. Write three for this one, at least one of which is not technical. *Hint: what happens to a platform decision when the vendor is acquired, and what did Sec. 12.3.2 say determines the cost of that day?*

12.10 Posed problem: the platform evaluation

No solution is given.

The situation. The water utility of Chapters 9, 10 and 11 has 400 substations, a rebuilt data layer, a fault detector with a poor signal ratio, and a board that has approved budget for "a digital twin platform". Three options are on the table:

- **Option A.** A large industrial-automation vendor's twin platform. Strong connector coverage for two of the three meter vendors, hosted store, built-in dashboards, an annual licence per substation, and a data model that is "flexible but structured".
- **Option B.** An open-source twin framework, self-hosted, with a message broker at its centre and a small community. Everything is configurable and nothing is supported.
- **Option C.** Continue assembling: the current stack, plus the six months of work Chapters 10 and 11 identified.

The utility has one team of four, no on-call rotation, a five-year horizon, and a regulator who has already asked for evidence once.

Produce an evaluation of no more than four pages containing:

1. **The component checklist** (Sec. 12.2.1) for each option, at the three levels of Sec. 12.2.2. Say for each option which component it helps *least* with, and check that against Chapter 3 Sec. 3.4's claim about where the value sits.
2. **The seven properties** (Sec. 12.3.1) answered in prose for each option, with one sentence per row. **Property 4 deserves special attention given the regulator**; say what you would require in writing.
3. **The export test** (Sec. 12.3.2) as you would actually run it, with what you would export and what you would check survived.
4. **The cost model** (Sec. 12.5.1) with all five parameters estimated and justified, computed at N = 400. State which parameter you are least sure of and what measurement would settle it.
5. **An honest estimate of `a`** for this team, based on evidence rather than intention. *What did their last piece of shared infrastructure actually cost the second time it was used?*
6. **The topology question** (Sec. 12.7.2) given three meter vendors, one of which is read manually every three months.
7. **A recommendation with expiry triggers**, per Sec. 12.8.4, including at least one organisational trigger.
8. **One paragraph on what you would tell the board**, in Chapter 1's units rather than in engineering ones.

What a good answer looks like. It runs the export test before comparing features, because that test settles more properties than any question. It notices that option A's per-substation licence at 400 substations is a *different shape* of cost from options B and C, and that the cost model's `p` term is where that shows up. It is honest that option B's "nothing is supported" is a real cost that lands on a team of four with no on-call rotation, and that Sec. 12.5.4's operations line may dominate everything else. It gives an evidence-based `a` rather than an aspirational one. And it says plainly that the regulator's interest makes property 4 a veto rather than a criterion – so an option failing it is eliminated before the arithmetic runs, not scored down within it.

12.11 Summary

Seven things, tied to the five objectives.

1. **Four chapters deferred here and the deferral was right** (Sec. 12.1). Each of them produced a durable list by refusing to produce a product list. A product table would date, would flatter the feature axis, and – worst – would substitute shopping for evaluation.
2. **Count what a platform supplies in Chapter 3's seven components, at three levels** (Sec. 12.2): hosts, implements, opinionates. Platforms cluster where every twin is the same and thin out at the estimator, the diagnose service and the command path – **which is exactly where Chapter 3 said the value is.** *(Objective 1.)*
3. **Evaluate by what your twin breaks without** (Sec. 12.3): event time preserved, absence representable, context with history, provenance out, datasets reproducible, model versions first-class, an enforcement point at the boundary. **A missing component you can build; a missing property you often cannot.** *(Objective 2.)*
4. **The exit question is the best single question** (Sec. 12.3.2): can you get your data out, with its provenance, without the vendor's help? It is testable in an afternoon, it subsumes five of the seven properties, and it fixes the cost of a reversal on the day you choose rather than the day you leave. *(Objective 2.)*
5. **Five parameters decide build-versus-buy** (Sec. 12.5), seeded from Chapter 1's own thirty-engineer-day estimate. For the demonstrator the crossover sits between the first twin and the second – too close to call at one, platform ahead at two – which explains why vendors sell to fleets and one-off projects hand-build. *(Objective 3.)*
6. **Two quantities decide it and they do different jobs** (Sec. 12.5.3). `rB - F` is the platform's head start at the first twin; `a - p` is the rate its lead grows per extra twin. **A platform with no head start needs `p < a` to ever pay off; one with a large head start can stay ahead across any fleet you will actually build even if `p > a`.** A vendor will argue about `F` and `r`, which set the head start and stop mattering as the fleet grows. `p` is measurable during the evaluation; `a` is a fact about your team, and **most teams overestimate it – the measured version is what your last project's second instance actually cost.** *(Objective 3.)*
7. **Composability is what makes `a` small, and its four seams were already built** (Sec. 12.6): the tuple between connector and store, the model interface, the state interface, and the context boundary that decides whether twin number two is a row or a fork. Every one was specified by an earlier chapter for a different reason – **which is why retrofitting composability is expensive, and why building the seams for one twin is cheap.** *(Objectives 4, 5.)*

And the note this chapter adds to Part III. Chapter 11 ended by observing that three chapters had reached the same conclusion from three directions: the question is never whether the thing would work, but whether a decision needs it. This chapter reaches it a fourth time, from the purchasing side, and adds the uncomfortable corollary: **the most expensive platform decisions are made by teams who never wrote down what they were buying it instead of.**

12.12 Exercises

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

12.12.1 *(Objective 1.)* A vendor says their platform "supports time-series data, asset modelling, and simulation". Rewrite that claim as Chapter 3 components at Sec. 12.2.2's three levels, listing the questions you would ask to place each one.

12.12.2 *(Objective 1.)* Which of Chapter 3's seven components would you be most suspicious of a platform supplying, and why? Give the two questions you would ask before believing it.

12.12.3 *(Objective 2.)* For each, name which of Sec. 12.3.1's seven properties is at stake and what breaks: (a) the platform stamps every reading with its arrival time and offers a separate optional "source timestamp" field; (b) missing readings are represented as gaps in the series, with no row; (c) the asset table has an "edit" button; (d) you can export data but the export omits which model version produced each estimate.

12.12.4 *(Objective 2.)* Design the export test of Sec. 12.3.2 for the demonstrator concretely: what would you export, into what, and what four things would you check survived? Then say which of the seven properties each check settles.

12.12.5 *(Objective 3.)* For a twin with $\texttt{B} = 80$ engineer-days, $\texttt{F} = 25$, $\texttt{r} = 0.5$, $\texttt{a} = 0.35$, $\texttt{p} = 0.2$: compute the head start and the slope, say which option is cheaper at $N = 1$ and whether that ever reverses, and state which single parameter you would most want to measure before trusting your answer.

12.12.6 *(Objective 3.)* Your team argues $\texttt{a} = 0.2$ because "we always build things properly". Name three pieces of evidence that would support or refute that, all of which are available from work already done.

12.12.7 *(Objective 4.)* Sec. 12.6.2's fourth seam decides whether the second twin is a row or a fork. For the demonstrator, list five things that must be configuration rather than code for pot 7's twin to be a row, and one thing that will probably still require code.

12.12.8 *(Objective 4.)* A colleague extracts the connector into a shared library used by three twins. Six months later it has a `if twin_name == ...` in four places. Diagnose which of Sec. 12.6.3's three failures occurred, and give the fix that does not involve deleting the special cases.

12.12.9 *(Objective 5.)* A platform requires its connector to run in its cloud. Work through what that costs the offshore turbine, using Chapter 9's volume arithmetic and Chapter 3 Sec. 3.6's three forces, and say whether it is a veto or a cost.

12.12.10 *(Objectives 1-5, and the one that leaves the book.)* Pick one real twin platform – any one, commercial or open-source – and run Sec. 12.3.3's one-day evaluation against the demonstrator: the component checklist, the schema fit against Chapter 10 Sec. 10.9.2's eleven tables, the seven properties in prose, and the export test. Write the one-page output. Then note which of the four activities told you the most, and whether it was the one you expected.

Solutions and hints

12.12.1. "Time-series data" is the store, and the questions are Sec. 12.3.1's properties 1, 2 and 3 – does it keep event time, can it represent absence with a reason, can it answer as-of questions. "Asset modelling" is context, and the question is whether it has validity intervals or an edit button (Chapter 10 Sec. 10.4.3). "Simulation" is the runner, and the questions are which model interfaces it accepts (Chapter 6's FMI discussion) and whether two versions can run against the same scenario (property 6). **Note that the claim named three components and said nothing about the level**, which is the point of the exercise: "supports" is unanswerable until you ask hosts, implements or opinionates.

12.12.2. The command path and actuation guard. Two questions: (i) where does *my* fail-safe logic run, per command, and can I express "refuse if the twin state is older than N minutes" (Chapter 3 Sec. 3.2.7 and Chapter 11 Sec. 11.6.4)? (ii) what is audited with each command, and can I get that record out (property 4)? A platform that supplies a command path but not per-command fail-safe has supplied the cheap half of the most expensive cell in Chapter 3's grid. A defensible second answer is the diagnose service, on the grounds that a supplied one encodes somebody else's value metric.

12.12.3. (a) Property 1. The optional field is the tell: optional means some path will not populate it, and Chapter 10 Sec. 10.2.5 pinned `event_time` precisely because two names for one concept produce half-populated columns. Breaks every Chapter 7 comparison, silently. (b) Property 2. Breaks Chapter 11's third outcome, Chapter 9's completeness count, and the distinction between "no water" and "no reading". (c) Property 3 – an edit button means update-in-place, so history is destroyed, and Chapter 10 Sec. 10.4.2's whole argument applies. (d) Property 4, and note that it also fails the exit question, which is why Sec. 12.3.2 says the export test settles several properties at once.

12.12.4. Export a month of: measurements for one binding, the context rows for that binding, the twin states and the alerts. Load them into a plain database. Check: (i) do measurement rows carry an event time distinct from ingest time – property 1; (ii) are failed polls present as rows with a quality, not absent – property 2; (iii) does the context export include `valid_from/valid_to` or only the current row – property 3; (iv) can you join an alert back to the model version, parameter set and input watermark that produced it – property 4. **A fifth check worth adding: re-run the same export next week and see whether you get the same rows for the same period** – property 5, and it catches a store that has silently absorbed late data (Chapter 10 Sec. 10.7.2).

12.12.5. `Assemble(N) = 80(1 + 0.35(N-1)) = 80 + 28(N-1)`. `Platform(N) = 25 + 80(0.5) + 80(0.2)(N-1) = 65 + 16(N-1)`. At N = 1: 80 versus 65, **platform already ahead** – because `F` is small relative to `B` and `r` is high. The gap only widens. The parameter to measure: `r`, because at `B` = 80 the platform's advantage at N = 1 is entirely `rB - F = 40 - 25 = 15` days, and `r` is the softest number in the model – it is a vendor's claim until you have built something on it. If `r` is really 0.3, the platform's advantage at N = 1 disappears (`24 - 25 = -1`).

12.12.6. (i) The actual effort logged for the second instance of your last shared component, against the first – that ratio *is* `a`, measured. (ii) Whether that component currently contains conditionals naming specific instances (12.12.8's symptom); if it does, the reuse was nominal. (iii) Who is on record as owning it, and whether that person still works on it (Sec. 12.6.3's third failure). **All three are available from work already done,**

which is the point: `a` is measurable and teams argue about it as though it were a matter of character.

12.12.7. Configuration: the binding (which pot maps to which sensor and multiplexer channel); the parameter set (`k_night`, `k_day`, `g` differ per pot); the schedule and doses; the alert threshold and its spread; the validity envelope bounds. Probably still code: **anything where pot 7 is physically different in kind** – a different sensor model with a different response curve, which is a new decoder in Chapter 9's handler and therefore a code change, not a row. That is the right boundary: configuration covers different *values*, code covers different *kinds*.

12.12.8. Sec. 12.6.3's **second** failure primarily – the parts were extracted and the glue was not, so per-twin wiring leaked back into the shared part as conditionals. The third failure is usually alongside it: a library with three consumers and no owner accumulates exactly this. **The fix that is not deletion:** turn each conditional into a *seam* – a configuration value or an injected strategy, per Sec. 12.6.2 – so the variation lives at the boundary rather than inside the part, and then give the part an owner. Deleting the special cases without doing that just breaks three twins.

12.12.9. Chapter 9's arithmetic: 2.3 TB per turbine per year of raw vibration, so the raw stream cannot cross a satellite or cellular link at all – which means feature extraction must run at the turbine regardless of what the platform wants. Chapter 3 Sec. 3.6's forces: the link is intermittent (force 2) and buffering must therefore be local (Chapter 9 Sec. 9.8.2). **So it is a veto, not a cost** – the platform cannot host the connector because the data physically cannot get to it. The subtler answer, and the better one, is that it may be a partial veto: the platform could host a *second-stage* connector consuming already-extracted features, with your own first-stage at the edge, which is Chapter 3's per-component rule saving the deal.

12.12.10. No solution. One prediction: the export test will tell you the most and it is the activity most likely to be skipped, because it requires credentials and half a day rather than a conversation. The component checklist will feel most productive and will change your mind least.

12.13 Where to go next

In this book. Chapter 13 covers the standards, which is where several of this chapter's questions get portable answers: property 4's provenance and the exit question's interoperability both improve when there is an agreed format to export *into*, and the asset-model and information-model work is the part that makes Sec. 12.6.2's fourth seam transferable between organisations rather than only within one [5], [6]. **Chapter 14 owns everything Sec. 12.5.4's operations line waved at**: running the thing, upgrading it, and the day a platform's version bump changes a number. Chapter 15 is where Sec. 12.5.2's `N` stops being a parameter and becomes the subject – ecosystems, twin-of-twins, and what composability means across organisational boundaries rather than within a team.

In the literature, if you want more.

- *The evaluation problem itself:* [1] is the source of this chapter's licence to supply a method rather than a survey, and is candid that existing evaluation approaches rest on subjective opinion and general-purpose criteria.
- *What is actually out there, evaluated properly:* [2] is the one to read – a survey of open-source twin frameworks conducted by implementing **the same case study across each of them**, which is the incubator this book has cited in Chapters 7, 8 and 11 [13], [14]. Reading the same twin built five ways is worth more than any feature table.
- *The platform view:* [4] proposes assembling twins out of reusable parts and delivering them as a service, and is Chapter 3 Sec. 3.5.4's source; [7] is a cloud-hosted instance of the same idea in construction; [15] and [16] show a working open framework built around a broker, including simulation and learned models side by side.
- *Composability and scale:* [8] argues composable and lean twins as the route to enterprise scale, which is Sec. 12.6's thesis stated from the business side; [12] on what breaks when twins are produced industrially rather than one at a time.
- *Topology:* [9] and [3] on placement across the edge-to-cloud continuum as the design variable, and the argument that fragmentation is the obstacle; [17] codifies the recurring deployment patterns.
- *Consulted, not drawn on above:* [18] on how platforms present twins – the reference Chapter 2 parked for this chapter – [19] and [20] on further framework comparisons, [21] on twins as microservices, and [22] on the platform properties a buyer should be able to measure.

In the demonstrator. Exercise 12.12.10 is the assignment and it is the cheapest useful thing in this chapter: pick one real platform, spend a day, write the page. Even if you never buy anything, you will have Sec. 12.3.1's seven questions answered for one real system, and the next evaluation takes half as long. **And run the export test even on the stack you assembled yourself** – Sec. 12.3.2's question is not only for vendors, and a home-built store that fails it has locked you in to yourself.

12.14 References

[1] Z. Tang, D. Zhuang, and J. Zhang, "Evaluation framework for domain-specific digital twin platforms," *Scientific Reports*, vol. 15, no. 1, p. 10544, Mar. 2025, doi: 10.1038/s41598-024-82154-8.

[2] 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.

[3] A. Barbone, S. Burattini, M. Martinelli, M. Picone, A. Ricci, and A. Virdis, "Digital Twin Continuum: A Key Enabler for Pervasive Cyber-Physical Environments," in *2024 33rd International Conference on Computer Communications and Networks (ICCCN)*, Jul. 2024, pp. 1–9. doi: 10.1109/ICCCN61486.2024.10637565.

[4] 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.

[5] A. C. Marosi *et al.*, "Interoperable Data Analytics Reference Architectures Empowering Digital-Twin-Aided Manufacturing," *Future Internet*, vol. 14, no. 4, p. 114, Apr. 2022, doi: 10.3390/fi14040114.

[6] 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.

[7] 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.

[8] P. van Schalkwyk and D. Isaacs, "Achieving Scale Through Composable and Lean Digital Twins," in *The Digital Twin*, N. Crespi, A. T. Drobot, and R. Minerva, Eds., Cham: Springer International Publishing, 2023, pp. 153–180. doi: 10.1007/978-3-031-21343-4_6.

[9] 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.

[10] A. Barbone, N. Bicocchi, M. Martinelli, R. Morandi, and M. Picone, "On-device AI and digital twins: A synergistic approach to intelligent cyber-physical systems," *Future Generation Computer Systems*, vol. 175, p. 108068, Feb. 2026, doi: 10.1016/j.future.2025.108068.

[11] Y. Lu, X. Huang, K. Zhang, S. Maharjan, and Y. Zhang, "Communication-Efficient Federated Learning and Permissioned Blockchain for Digital Twin Edge Networks," *IEEE Internet of Things Journal*, vol. 8, no. 4, pp. 2276–2288, Feb. 2021, doi: 10.1109/JIOT.2020.3015772.

[12] S. A. Niederer, M. S. Sacks, M. Girolami, and K. Willcox, "Scaling digital twins from the artisanal to the industrial," *Nature Computational Science*, vol. 1, no. 5, pp. 313–320, May 2021, doi: 10.1038/s43588-021-00072-5.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] M. Jacoby *et al.*, "Open-Source Implementations of the Reactive Asset Administration Shell: A Survey," *Sensors*, vol. 23, p. 5229, May 2023, doi: 10.3390/s23115229.

[20] B. Caesar, K. Barton, D. Tilbury, and A. Fay, "Digital Twin Framework for Reconfiguration Management: Concept & Evaluation," *IEEE Access*, vol. PP, pp. 1–1, Jan. 2023, doi: 10.1109/ACCESS.2023.3331221.

[21] A. G. Wermann and J. A. Wickboldt, "KTWIN: A Serverless Kubernetes-based Digital Twin Platform." arXiv, Aug. 2024. doi: 10.48550/arXiv.2408.01635.

[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.