Chapter 10 – Data Engineering for Twins: Time Series, Context, and Provenance

10.0 Before you start

Where we are. Chapter 9 built the connector and stopped, deliberately, at the moment a tuple was accepted. It said what had to be recorded – the tuple, two timestamps, a quality state, completeness counts, a watermark – and refused to say where any of it lives. This chapter says where.

It is also the chapter with the oldest outstanding debt in the book. Chapter 3 Sec. 3.2.2 described the store, listed three kinds of content, posed two questions the store must answer, and then did something unusual: it told you what *this* chapter would say. "Chapter 10 calls this the part everyone underestimates, and it is right: a value without its context is not data, it is a number." Sec. 10.1 makes good on the sentence Chapter 3 wrote for it.

The register, unchanged from Chapter 9 and worth restating. You have designed schemas, chosen databases and argued about normalisation. This chapter teaches none of that.

> **This chapter does not teach data engineering. It teaches the four things a twin's data layer must do that an ordinary application's does not, and it is honest that the rest is work you already know how to do.**

The four:

1. **Represent absence as richly as presence.** Chapter 9 produced six quality states, two of which carry no value. A schema that collapses them to null has destroyed the distinction Chapter 3 defended and Chapter 6's estimator consumes.
2. **Give context a history.** The pot was repotted, the sensor was replaced, the pump was recalibrated. A context table that only knows the present cannot interpret last month's measurements.
3. **Answer both of Chapter 3's questions.** What was measured at time t is the tractable one. What the twin *believed* at time t is the one that decides whether an incident review is possible, and it needs a timestamp Chapter 3's tuple does not have.
4. **Make a dataset nameable.** Chapter 7 said "week 3" and Chapter 8 said "the training set" as though those phrases identified something. Making them identify something is this chapter's job.

What you are assumed to know. Everything so far. Especially: Chapter 3's store, binding, the two questions, and the retain-the-derived-output rule; Chapter 7's calibration windows, hold-out, credibility argument, and the five incident-review questions of Sec. 7.7.3; Chapter 8's three model artifacts and its retroactive-invalidation expiry trigger; and all of Chapter 9 – the tuple, event and ingest time, the six quality states, completeness, and the watermark.

The maths budget. As Chapter 9: more arithmetic welcome, no new mathematics. Two set pieces, both arithmetic – a four-query walk through one incident (Sec. 10.5.3),

and retention sizing carried from Chapter 9's volume figures (Sec. 10.3.5).

What this chapter deliberately does not cover. Database technology or product selection – no engine is named, and Chapter 12 selects. Ontology formalisms – asset models and knowledge graphs are named for the *problem* they solve and handed to Chapter 13. Cross-organisation data sharing and data spaces – Chapter 15. The Findable, Accessible, Interoperable and Reusable (FAIR) principles, and data standards – Chapter 13. Feature stores as a technology – named once. Ingest – Chapter 9. Query APIs and services – Chapter 11. Backup and operational running – Chapter 14. Security and access control – Chapter 3 Sec. 3.3.3 and Chapter 13.

Learning objectives

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

1. **Distinguish** the three kinds of content a twin stores, **state** the shape and access pattern of each, and **decide** what they must share.
2. **Design** a measurement schema that represents Chapter 9's six quality states without collapsing absence to null, and **avoid** the series-cardinality trap.
3. **Model** context as something with a history, **express** a change to the physical twin as an event with a validity interval, and **join** a measurement to the context that was true when it was taken.
4. **Answer** both of Chapter 3's questions, and **explain** why the second needs a third timestamp that measurements do not.
5. **Specify** a reproducible dataset and a provenance chain sufficient for Chapter 7's incident review and Chapter 8's retraining.

10.1 The part everyone underestimates

Chapter 3 promised this sentence, so here it is, in full:

> **A value without its context is not data, it is a number.**

`606` is not a moisture reading. It becomes one when you know which pot, which sensor, in which units, at what moment, under which calibration, with what confidence, and whether the plant had been repotted that morning. Everything in this chapter is an elaboration of that sentence, and the reason it is underestimated is that **the number arrives first and the context arrives never, unless somebody makes it.**

Three reasons the data layer is estimated at a quarter of its cost.

The measurement part looks like the whole job, and it is the tractable quarter. Appending timestamped values to a store is a solved problem with good tooling. Recording that pot 3's sensor was replaced on 14 August and that every reading before then came from a different device is not solved by any tool, because no tool knows that it happened.

The context is small, so it feels unimportant. Chapter 3 already characterised it correctly: low volume, mutable, relational. Twelve pots and their sensors is a few dozen rows against millions of measurements. **Volume is a terrible proxy for consequence**, and this is the clearest example in the book: the few dozen rows decide what the millions mean.

The parts that pay off are the parts you cannot add later. You can add an index next year. You cannot add last year's calibration record, or the fact that the twin believed something different at the time it alerted. **Every requirement in this chapter that matters is one that is free to satisfy today and impossible to satisfy retroactively**, which is precisely the category of requirement that loses arguments about scope.

And a warning about how this failure presents. It does not present as a data problem. It presents as a modelling problem. Chapter 7 built its whole machinery on residuals, and a residual computed against a measurement whose context is wrong is a number that will be attributed to an assumption row, argued about in a review, and eventually fixed with a parameter. Chapter 9 made the same point about aliasing. This is the second time the book has found a modelling symptom with an infrastructure cause, and it will not be the last.

10.2 Three kinds of content, three shapes

Chapter 3 Sec. 3.2.2 named them. Here is what each one implies for a schema.

10.2.1 Measurements

Shape. High volume, append-only, time-indexed, immutable once correct. Chapter 9's tuple, arriving at `Ts`.

Access patterns, and they are narrower than a general-purpose store assumes:

- *Range scan by series and time.* "Moisture for pot 3, 09:00 to 18:00 on 14 August." This is nearly every read the twin does.
- *Latest value per series.* The state estimator's input, and the age-of-twin metric.
- *Aggregate over a window.* Hourly means for a dashboard, daily completeness counts.

What it does not need. Update in place. Joins across series in the storage engine. Transactions spanning many series. Recognising that is what makes a time-series store the right shape rather than a fashion.

10.2.2 Context

Shape. Low volume, mutable, relational – and, as Sec. 10.4 argues, not actually mutable at all once you take history seriously.

What is in it, for the demonstrator. Pot, plant, sensor, pump, multiplexer port, relay coil, pump calibration, model version, parameter set, and the bindings tying them together. Chapter 3 Sec. 3.7.2 listed most of these and Chapter 9 added the multiplexer channel.

Access patterns. Point lookups by binding, and – the one that distinguishes a twin – *point lookups by binding as of a past moment.*

10.2.3 Derived output

Shape. Moderate volume, append-only, and **the kind most often not stored at all**, which is the mistake Chapter 3 Sec. 3.7.2 flagged: retain the derived output, because

re-running today's model against August's measurements answers a different question.

What is in it. Twin state with its `as_of`, residuals, forecasts, uncertainty ranges, alerts, and the record of what each service decided and on what basis.

Why it is separate from measurements even though it looks similar. Chapter 5 Sec. 5.2.4 already insisted that predicted and measured belong in different columns, and warned that a twin storing both in one table with a flag will eventually plot one as the other. The schema-level version of that warning: **derived output carries a timestamp measurements do not have** (Sec. 10.5.2), so it does not fit the measurement schema even if you force it to.

10.2.4 One binding, not one engine

Chapter 3's design note was that the three need not live in one engine and usually should not, but must share the binding so a context row and a measurement row can be joined. That instinct is the common pattern: a dual-core architecture combining a relational database with a time-series database is proposed precisely to manage the differing shapes of twin data [1].

What "share the binding" actually requires, stated as three rules because it is where the idea usually fails in practice:

1. **The binding is a string with a grammar, decided once.** `pot-3/moisture` is a binding. `pot 3` in one table and `Pot3` in another is not a shared key, it is a future incident.
2. **Nothing else is a join key.** Not the sensor serial number, not the multiplexer channel, not the row id. Those are attributes *of* a binding at a moment in time, and Sec. 10.3.2 shows what goes wrong when they leak into keys.
3. **The join across engines happens in your code, so its correctness is your problem.** No foreign key will enforce it. A daily job that checks every distinct binding in the measurement store against the context store and reports orphans costs an hour to write and finds real defects.

10.2.5 Pinning two names, because Chapter 3 gave each of them twice

A small thing that will otherwise cost a reader an afternoon. Chapter 3 described the measurement tuple in two places and used different spellings:

Concept	Ch3 Sec. 3.2.1	Ch3 Sec. 3.7.2	Ch9	**Pinned here**
What is measured	`quantity`	`parameter`	`quantity`	**`quantity`**
When it was true	`timestamp`	`measured_at`	*event time*	**`event_time`**

From here to the end of the book: `quantity` and `event_time`. The concepts were always the same; only the spelling drifted, and drifting spellings in a schema are how two teams build two half-compatible stores.

And the general rule this is an instance of. Column names in a twin outlive the code that writes them, because the data outlives the code. Fix them early, write them down where a reviser will find them, and treat a rename as the migration it is.

10.3 Storing measurements

10.3.1 What a time-series store buys, in one paragraph

You know this. Briefly, so the rest of the section has a footing: a time-series store exploits the fact that rows arrive in time order, are never updated, and are read as ranges within a series – which buys compression that general-purpose engines cannot match, and range scans that do not touch an index per row. Retention and downsampling are usually built in. **None of that is twin-specific**, and none of it is why a twin's measurement store is hard. The rest of this section is the part that is.

10.3.2 Series identity, and the cardinality trap

A **series key** is the set of fields identifying one time series. Choosing it is the one storage decision that is expensive to reverse.

The right key for the demonstrator is `(binding, quantity)`: `pot-3/moisture`, `greenhouse/air_temperature`, and so on. Twelve pots plus a greenhouse gives twenty-two series (Chapter 9 Sec. 9.8.1), and that number changes only when the greenhouse does.

The trap. It is tempting to put more in the key, because more identity seems safer:

```
(binding, quantity, sensor_serial, firmware_version, calibration_id)
```

Every one of those additions is something that *changes*. Replace a sensor and pot 3's moisture becomes a new series. Now:

- a range scan over "moisture for pot 3 in 2026" must union an unknown number of series;
- every dashboard, alert and calibration query needs to know about the replacement;
- the series count grows monotonically with maintenance activity, forever;
- and Chapter 7's continuity of residuals across the change is invisible rather than merely difficult.

  The rule. The series key contains what identifies the *measured thing*. Everything that identifies the *measuring apparatus* belongs in context, joined by time (Sec. 10.4.5). Chapter 3's binding was designed for exactly this: it maps a stream to a physical twin, and it deliberately does not name the hardware.

The objection, and it is a fair one. "But I need to know which sensor produced this reading." Yes – and you get it from the context store, by asking which sensor was bound to `pot-3/moisture` at that `event_time`. That is one join, it is correct across replacements, and it is the only arrangement in which "compare the sensor's readings before and after the swap" is a query rather than a project.

10.3.3 Representing absence, which is where Chapter 9 either survives or dies

Chapter 9 Sec. 9.6.1 defined six quality states. Two of them – `no_reading` and `not_attempted` – carry **no value at all**, and one – `substituted` – carries a value that was never measured. Chapter 3 defended this in principle: a reading you are unsure of and a reading you did not get are different, and mapping both to null destroys information the estimator needed.

Here is where that principle meets a schema, and where it usually dies. The natural table is `(series, event_time, value)`. A missing reading is just an absent row. A `substituted` value looks identical to a measured one. And a nullable `value` column collapses `no_reading` and `not_attempted` into the same thing.

The schema that works is not clever, it is only deliberate:

```
measurement(
    binding, quantity,        -- series key
    event_time,               -- pinned name, Sec. 10.2.5
    value        NULLABLE,    -- absent for no_reading / not_attempted
    unit,
    quality,                  -- one of the six, NOT NULL
    ingest_time,              -- Ch9 Sec. 9.5.1; operational, not physical
    PRIMARY KEY (binding, quantity, event_time)
)
```

Three decisions in that block, each of which is routinely got wrong.

`quality` is not nullable and has no default. Every row states its quality explicitly. A default of `good` means that the day somebody inserts rows by a path nobody thought about, those rows claim to be measurements.

Rows exist for failures. A failed poll writes a row with a null `value` and `quality = no_reading`. This is the decision that surprises people, because it stores non-data – and it is what makes Sec. 10.3.4's completeness check a query rather than a reconstruction, and what makes "was this gap a failure or a scheduled outage?" answerable in November.

`substituted` is a value with a quality, never a value alone. Chapter 8 Sec. 8.5.1 showed what a training set learns from unlabelled bad data; substituted values are the same hazard with a friendlier face. Chapter 9's table said `substituted` must be excluded from calibration, hold-out and training. **That exclusion is a `WHERE` clause, and it only exists if the column does.**

> **The test for this section.** Write the query "give me pot 3's moisture for week 3, using only genuine measurements". If it is one predicate on `quality`, the schema is right. If it requires knowing which rows were filled in, and that knowledge lives in somebody's memory, the schema has already failed and no amount of downstream care will recover it.

10.3.4 Completeness, stored rather than recomputed

Chapter 9 Sec. 9.7.1 defined completeness – received over expected – and set the demonstrator's rule: 144 expected per binding per day, mark a window degraded below

90 per cent, and exclude degraded windows from calibration and validation. It then said explicitly that where this lives is Chapter 10's problem.

It lives in a small table, computed once per window, not recomputed on demand.

```
window_quality(
    binding, quantity,
    window_start, window_end,
    expected_count, received_count, good_count,
    completeness,             -- good_count / expected_count
    degraded,                 -- boolean, from the threshold
    computed_at
)
```

Three reasons to materialise it rather than compute it in the query that needs it.

It is the thing Sec. 10.7's dataset definition points at. An exclusion list that recomputes its own criteria is not an exclusion list.

The threshold is a decision with a date. If the 90 per cent rule changes to 95, you want to know which datasets were built under which rule. A stored `degraded` flag with a `computed_at` gives you that; a predicate evaluated at query time silently rewrites history.

It is cheap and it is the first place anybody looks. Twenty-two series times 365 days is eight thousand rows a year.

Note the three counts, which are not the same number. `expected_count` comes from `Ts` and the window. `received_count` includes `no_reading` rows, because those are received facts about the world. `good_count` counts only usable measurements. Completeness is computed from the third against the first – and reporting the second as though it were the third is a real and common way to make a broken sensor look like a healthy one.

10.3.5 Retention, which is a correctness decision here and a cost decision elsewhere

Chapter 9 Sec. 9.8.1 did the volume arithmetic. Carrying it forward:

The demonstrator. About 139 MB per year of raw measurements. Derived output adds: twin state every 10 minutes per pot (12 x 144 = 1,728 rows a day), a residual per dose (24 a day), and alerts, which are rare. Call it another 80 MB a year, generously. So:

```
raw measurements, 2 years at full resolution   = about 278 MB
derived output, 2 years                        = about 160 MB
hourly rollups thereafter, 22 x 8,760 per year = 192,720 rows = about 23 MB/year
ten further years of rollups                   = about 230 MB
```

Under a gigabyte for a decade. So the retention question for this twin is not "what can we afford to keep" – it is **"what does keeping less make impossible?"**, and that is a correctness question with three concrete answers:

- *Chapter 7's recalibration* needs several weeks of full-resolution history, and its expiry conditions are measured in weeks.
- *Chapter 8's learned correction* wants a year or more, because it must see the seasons it will be asked about.
- *Chapter 7's incident review* needs whatever period an argument might reach back over, which in a regulated setting is a legal answer rather than an engineering one.

And the trap that makes downsampling different from compression. An hourly mean is not a lossy version of the readings – it is a different quantity. Chapter 7's residual is the difference between a prediction and a measurement *at the dose time*; from hourly means you cannot compute it at all, because the step is inside the hour. **Downsample only after asking which of Chapter 7's numbers stops being recomputable**, and record the answer beside the retention policy.

The turbine, for contrast. Chapter 9's arithmetic gave 2.3 TB per turbine per year for vibration alone, and about 180 TB a year for an eighty-turbine farm. Now retention *is* a cost decision, and the answer that the structural-monitoring architectures converge on is: keep derived features indefinitely, keep raw windows only around events, and accept that a question nobody anticipated cannot be asked of the raw data later [2], [3].

> **The general shape.** Small twin: keep everything, and spend the thinking on context and provenance instead. Large twin: decide in advance which questions the raw data must be able to answer, because you are choosing now which future questions are unanswerable. Both are decisions; only one of them is about money.

10.4 Storing context

10.4.1 What context is, concretely

For the demonstrator, everything that turns `606` into a moisture reading:

Context	Example	Changes when
Asset	pot 3, in bay 2, planted with basil on 2026-06-01	Repotting, replanting, moving
Instrument	STEMMA sensor S-0147 on multiplexer channel 3	Replacement, rewiring
Actuator	pump P-0031 on relay coil 5, `slope = 0.02`, `offset = 1.5`	Replacement, recalibration
Binding	`pot-3/moisture` maps to (pot 3, sensor S-0147)	Any of the above
Model	`pot-3/v2`, parameters `k_night` = 3.67, `k_day` = 4.28, `g` = 47.0	Refit, model version change
Calibration	sensor last calibrated 2026-07-02	Recalibration
Envelope	valid for readings 350 to 900	Model or sensor change

Chapter 1's greenhouse citation already made the case that this is real work: keeping track of where a plant sits so it can be mapped to the correct sensor stream, and coping when plants are moved [4].

10.4.2 Context has a history, and that is the whole difficulty

Here is the sentence that this section exists for:

> **A context table that only knows the present cannot interpret the past, and a twin is a system whose entire value comes from interpreting the past.**

Every query in Chapter 7 reaches backwards. The calibration campaign fitted `k` from six nights in week 1. The hold-out tested week 3. The incident review in November asks about 14 August. If the context store answers those questions with *today's* pot, sensor and parameters, then every one of them is wrong in exactly the cases where it matters – the cases where something changed.

And Chapter 3 already required the fix, in two places that were about different things and are actually about this one. Sec. 3.3.1: binding is data, versioned, auditable, and a change to it is an event the twin records. Sec. 3.3.4: every change to the physical twin must be an event the twin can see, because a model fitted before a pump swap is not valid after it, and the only thing that can invalidate it is a record that the swap happened.

10.4.3 Validity intervals, worked

The mechanism is one pair of columns and one rule.

```
context_binding(
    binding,
    pot_id, sensor_id, mux_channel,
    valid_from,               -- inclusive
    valid_to,                 -- exclusive; open-ended for the current row
    changed_by, change_reason,
    PRIMARY KEY (binding, valid_from)
)
```

> **The rule: context rows are never updated in place.** A change closes the current row by setting its `valid_to`, and inserts a new row with `valid_from` equal to the same instant. The table only grows.

Worked: pot 3 is repotted on 14 August at 11:00, and its sensor is replaced at the same time.

binding	pot_id	sensor_id	mux_channel	valid_from	valid_to
`pot-3/moisture`	pot-3	S-0147	3	2026-06-01 09:00	2026-08-14 11:00
`pot-3/moisture`	pot-3	S-0203	3	2026-08-14 11:00	*(open)*

Now the query that matters. Which sensor produced the reading whose `event_time` is 14 August 09:40?

```
select sensor_id from context_binding
 where binding = 'pot-3/moisture'
   and valid_from <= '2026-08-14 09:40'
   and (valid_to is null or valid_to > '2026-08-14 09:40')
```

`S-0147`. And for a reading at 12:10, `S-0203`. **One join, correct across the change, and available for every past moment** – which is what Sec. 10.3.2's series-key rule bought.

Three consequences worth stating.

Chapter 7's parameters are context too. `k_day` = 4.28 was fitted on week 2 and is valid from some date. When Chapter 7's expiry fires and a refit happens, that is a new row, not an update – and "which parameters did the twin use on 14 August?" becomes the same query shape as the sensor one.

The repotting invalidates the parameters, and now something can say so. Chapter 4's assumption A4 said parameters hold for weeks and break on repotting; Chapter 7 made "the rig, plant or sensor changed" an expiry trigger and noted that the trigger fails in practice because nothing detects it. This table is where a human's action becomes a machine-readable event. It still requires the human to record it – but now there is somewhere to record it, which is the difference between a process that can work and one that cannot.

***`changed_by`** and **`change_reason`** are not decoration.* When Chapter 7's residuals jump on 14 August, the first useful question is what else happened on 14 August, and this table is the answer.

10.4.4 Change as an event, and the two things that must be true

Chapter 3's rule was that every change is an event the twin can see. Two properties make that rule work rather than merely sound good.

The change must have a time that is the *physical* time, not the recording time. Somebody repots at 11:00 and updates the system at 16:00. If `valid_from` is 16:00, five hours of readings are attributed to the old pot and the old sensor. **Record the time it happened, and accept that the row will be inserted late** – which is the same event-time-versus-ingest-time distinction Chapter 9 Sec. 9.5.1 drew for measurements, arriving in the context store.

The set of change types must be enumerated, and short. For the demonstrator: sensor replaced, pump replaced, pump recalibrated, plant repotted, plant replaced, pot moved, binding rewired, model version changed, parameters refitted. Nine. A free-text change log is not a machine-readable event and cannot trigger anything. Work on twin evolution distinguishes continuous change such as wear from discontinuous change such as component replacement [5], and it is the discontinuous kind that this table exists for – the kind that no model will explain and no residual monitor should be asked to.

10.4.5 Joining by time, not by key

Chapter 9 Sec. 9.9.2 created a problem and named it: the greenhouse's air temperature and humidity are one binding shared across all pots, so the store must join them to a pot's moisture **by time rather than by pot**.

Why this is not a normal join. There is no key relating `greenhouse/air_temperature` to `pot-3/moisture`. What relates them is that they were true at about the same moment – and "about" is doing work, because the two series are not guaranteed to have samples at the same instants even at the same `Ts`.

Three ways to do it, in increasing order of honesty.

Approach	How	When it is right
Nearest-value join	For each moisture row, take the greenhouse row closest in time	Quick, and fine when the joined quantity moves slowly relative to `Ts`
Bounded nearest	The same, but refuse if the nearest is more than `w` away	**The default.** Refusing is the point: a temperature two hours stale is not a temperature
Resample both to a grid	Interpolate both onto fixed instants	When many series must be aligned, e.g. building Chapter 8's training rows

Figure 10.2 shows why the middle row is the default rather than the first.

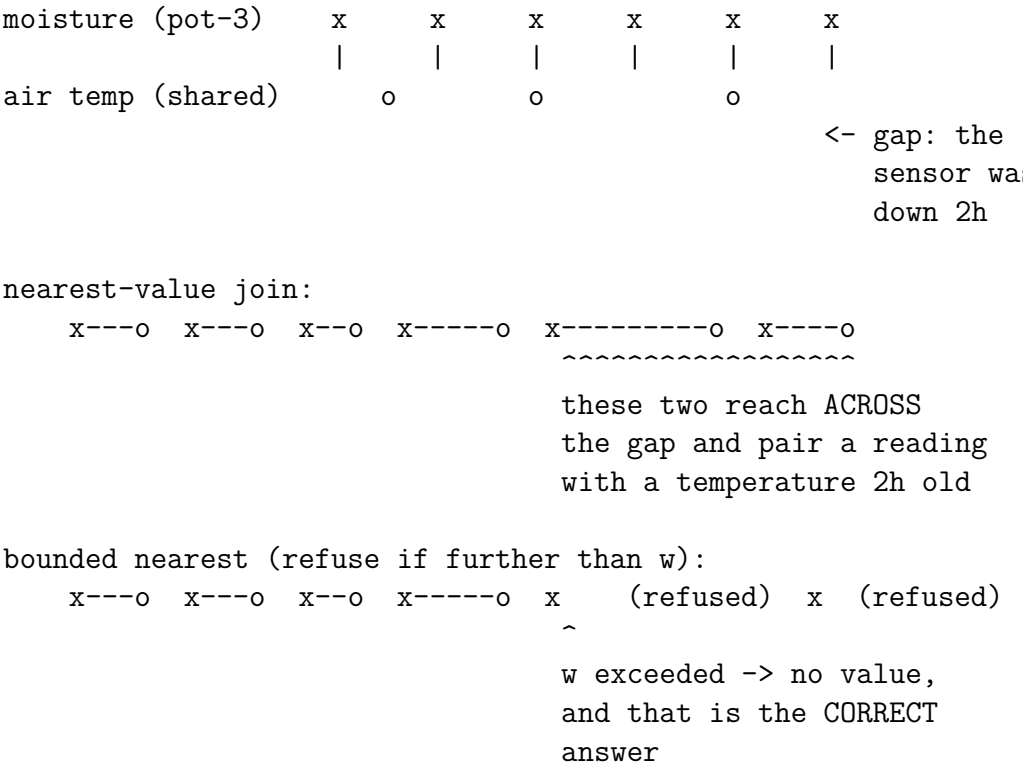

Figure 10.2 Why bounded nearest is the default. The unbounded

```
join always returns something, and what it returns
across a gap is fiction with a measured value's
type.
```

And the rule that keeps all three honest: a joined value that was interpolated or taken from a different instant is **derived**, not measured. Chapter 9's `substituted` state exists for exactly this, and the discipline is the same one Sec. 10.3.3 established – if the joined row feeds Chapter 7's calibration or Chapter 8's training, its provenance must say that its temperature column was matched from 40 seconds away, not measured there.

Pick `w` from the physics, not from taste. Greenhouse air temperature changes over tens of minutes, so `w` = 15 minutes is generous and safe. If you were joining a 2 kHz vibration channel to another, `w` would be microseconds and Chapter 9 Sec. 9.5.3's clock discussion would be the binding constraint.

10.4.6 Asset models and knowledge graphs, named

At some scale, the context store stops being a handful of tables and becomes a model of the physical system's structure: what is part of what, what is connected to what, what kind of thing each thing is. That representation has a name and a literature. Knowledge graphs are used to record the current structure of a physical system **and its changes**, and are a key technology for asset models in twin architectures – the greenhouse exemplar this book has cited since Chapter 1 is built that way, precisely because plants move [4].

What to take from that, and what to leave.

Take the problem statement. Structure is data, structure changes, and the changes are as important as the structure. That is Sec. 10.4.3 and Sec. 10.4.4, and it holds whether your context store is seven tables or a graph.

Take the scaling signal. When the number of *kinds* of relationship starts growing – pots in bays in greenhouses on sites, sensors on multiplexers on gateways, models composed of sub-models – a relational context store starts requiring a schema change for every new kind, and a graph does not. That is the moment to look, and not before.

Leave the formalism. Ontology languages, reasoners and their query semantics are a subject, and Chapter 13 covers the standards where asset models are specified. Nothing in this chapter needs them, and a twin whose context is seven well-maintained tables with validity intervals is ahead of most.

10.5 The two questions: bitemporality

This is the chapter's spine, and the section that most distinguishes a twin's data layer from an ordinary application's.

10.5.1 The two questions, restated

Chapter 3 Sec. 3.2.2:

"What was measured at time t?" and "what did the twin *believe* at time t?" A store that can only answer the first cannot support any post-hoc review of a decision the twin made.

Read carefully, the second question contains an ambiguity that turns out to be the whole thing. "What did the twin believe at time t" can mean:

- *What did the twin believe was true of the world at time t?* – answerable from today's data. This is a question about the world.
- *What did the twin believe, at time t, at that moment?* – answerable only if you recorded beliefs as they were held. This is a question about the system.

The second reading is the one an incident review needs, and it is the one that requires more than a measurement store.

10.5.2 Three times, not two

Chapter 9 gave measurements two times: `event_time` (when the world was that way) and `ingest_time` (when we learned). For derived output that is not enough, and here is why in one sentence: **a belief is about a moment and is held at a moment, and both matter.**

Timestamp	On	Means
`event_time`	measurements	when the world was in this state
`ingest_time`	measurements	when the connector accepted it
`as_of`	derived output	the moment this belief is *about*
`decision_time`	derived output	the moment the twin *formed* this belief

Chapter 3 Sec. 3.3.2 already required `as_of` to be an output rather than a log line, and Chapter 3 Sec. 3.7.3 put it in the estimator's output tuple. `decision_time` is the new one, and it is the one that makes the second question answerable.

Figure 10.1 draws the four on two axes, which is the picture that makes "what did we believe on 14 August?" a different question from "what was true on 14 August?".

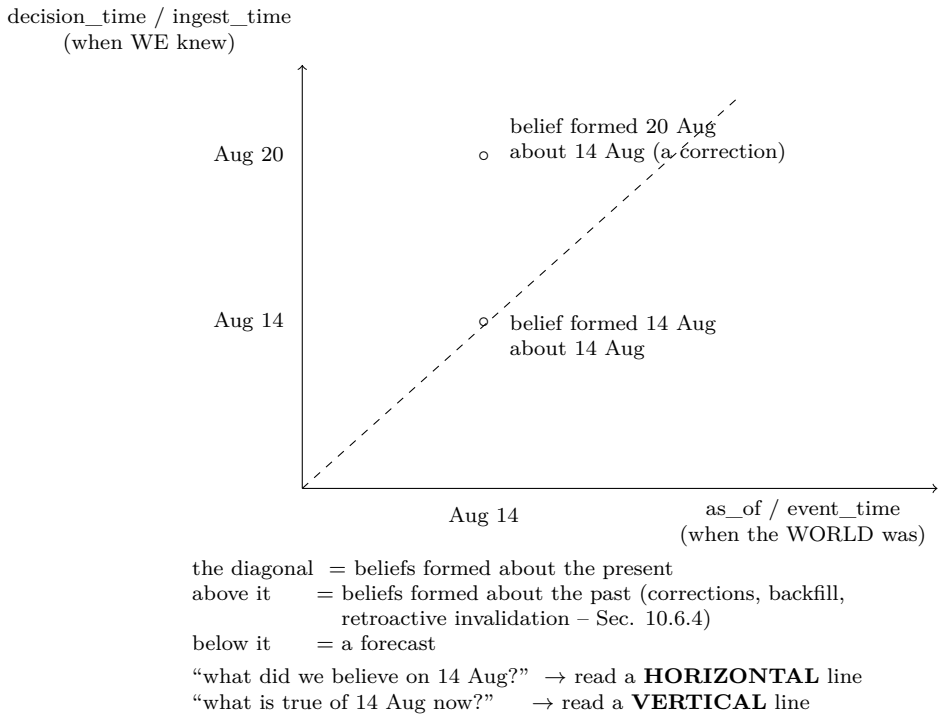

Figure 10.1 Two times, two questions. One timestamp collapses the plane to the diagonal and makes the horizontal question unanswerable.

The two reading directions in that figure are the whole of Sec. 10.5.3's worked alert, and they are why an audit ("why did you water then?") and a diagnosis ("what actually happened?") need different queries against the same table.

> **State it plainly, because it is a genuine amendment.** Chapter 3's measurement tuple was right for measurements and is insufficient for derived output. Measurements are bitemporal by accident – the world's time and ours. Derived output must be bitemporal **on purpose**, and the two pairs of times are not the same pair. Do not retrofit `decision_time` into the measurement tuple; the two content kinds have different temporal schemas because they are different kinds of thing.

So:

```
twin_state(
    binding, as_of, decision_time,
    moisture_estimate, uncertainty,
    model_version, parameter_set_version,
    inputs_watermark,          -- newest event_time this belief was allowed to see
    PRIMARY KEY (binding, as_of, decision_time)
)
```

`inputs_watermark` is the field that makes an incident review possible, and it is the one nobody adds unprompted. It records what the twin was allowed to know. Without it you can see what the twin believed but not whether it should have believed something else, which is the difference between a record and an explanation.

10.5.3 Worked: the alert on 14 August

All of Chapter 7's numbers apply: expected step from a 120 ml dose is `g` x 1.2 = 47.0 x 1.2 = **56.4** reading units, and the diagnose service alerts when the observed step is more than 9.0 units below that, i.e. below **47.4**.

What happened.

Wall clock	Event
17:00	Reading taken at the pot: 606. Ingested at 17:02
17:05	The controller commands a 120 ml dose. Watering event ingested at 17:07
17:10	Reading taken at the pot: 663. **The gateway's write is delayed, and the 17:20 reading with it; the connector's 17:12 and 17:22 polls return nothing after 17:00**
17:22	The diagnose service evaluates the 17:05 dose. It takes the last reading before the dose (17:00, value 606) and the newest reading available (also 17:00, value 606, because nothing later has arrived) and subtracts. Observed step: **0**. `0 < 47.4`, so it **alerts**: "dose did not land"
18:30	Backfill delivers the 17:10 reading, value 663, `event_time` 17:10, `ingest_time` 18:30
November	Somebody asks why the twin alerted on 14 August

Four queries, and two of them disagree.

#	Question	Answer	Needs
1	What was the moisture at 17:10?	**663**	The measurement store, keyed on `event_time`
2	What was the observed step for the 17:05 dose?	**663 - 606 = 57.0**, which is healthy: within one spread (3.0) of the expected 56.4	The measurement store
3	What did the twin believe the step was, when it decided at 17:22?	**0**	Derived output, with `decision_time`

#	Question	Answer	Needs
4	Was the alert justified given what was knowable at 17:22?	**Partly.** `inputs_watermark` was 17:00, so no post-dose reading existed and no correct step could be computed. But the service computed one anyway	`inputs_watermark`

Read rows 2 and 3 together. The same question – what was the observed step – has two correct answers, because it is two different questions wearing one sentence. Row 2 is about the world. Row 3 is about the system. A store that can only answer row 2 will tell the November reviewer that the step was 57.0 and the alert was therefore a bug, which is **false**, and somebody will spend a week looking for a defect in the diagnose service's threshold arithmetic that is not there.

And row 4 is the one that finds the real defects – two of them.

The first is timing. The service ran at 17:22 against a watermark of 17:00, which cannot contain a post-dose reading. Chapter 9 Sec. 9.7.3 allowed one poll interval of lateness; the diagnose service ignored that allowance.

The second is worse and only visible once you look at what it computed. The service subtracted its newest available reading from the pre-dose reading **without checking that the newest one was actually after the dose**. Both operands were the 17:00 reading, so the step came out as 0 – a number that looks like a measured absence of water and is in fact an arithmetic artifact of subtracting a reading from itself. **An undefined step was reported as a step of zero**, which is the same failure Chapter 9 Sec. 9.6.1 was designed to prevent one layer down: absence rendered as a value.

The fix is two lines in a different component, and it is Chapter 11's: require that the post-dose operand have an `event_time` after the dose, and evaluate a dose only when the measurement window is complete or the lateness allowance has expired. When it has expired with the window still incomplete, raise "cannot evaluate" rather than "dose did not land." Those are different alerts with different actions – one sends a technician to the rig, the other sends someone to look at the connector.

> **The point of this whole section.** That fix is invisible without `decision_time` and `inputs_watermark`. With them it took four queries and an afternoon. **The data layer did not prevent the incident; it made the incident diagnosable**, and that is the honest claim for bitemporality.

10.5.4 What it costs, and when to skip it

Bitemporality is not free, and pretending otherwise invites a reader to over-build.

What it costs. Two extra columns on derived output, a compound primary key, one more field the services must populate honestly, and a query habit: every question about

the past must say *as of when*. Storage cost is negligible; the real cost is that a service which lies about its `inputs_watermark` produces a record that is worse than none, because it is trusted.

Where you can skip it. Measurements, always – they are not beliefs. Dashboards showing current state. Anything whose output nobody will ever ask about retrospectively.

Where you cannot. Any output that fed a decision, which by Chapter 2's data-flow test is the whole of a twin as opposed to a shadow. Chapter 7's credibility argument depends on the incident review being possible, and Chapter 2 Sec. 2.8 established that the obligations change the moment the twin acts. **If the twin issues commands, `decision_time` is not optional and neither is retaining what it decided.**

10.6 Provenance: what produced this number

Chapter 7 Sec. 7.0 deferred "provenance plumbing" here, and Sec. 7.7.3 said "Chapter 10 does the data engineering properly". Here it is, and it is mostly bookkeeping that has to be decided rather than invented.

10.6.1 The chain

Provenance is the answer to "where did this number come from", followed recursively until you reach something physical. For the demonstrator's alert:

```
alert
 -> the residual it fired on
     -> the expected step  -> parameter set pot-3/v2  -> the calibration run
                                                          -> the six nights
                                                              -> measurements
     -> the observed step  -> two measurements
                              -> binding pot-3/moisture, valid at that time
                                  -> sensor S-0147
                                      -> its calibration record
 -> the model version that computed it
 -> the decision_time and inputs_watermark it ran under
```

Figure 10.3 The provenance chain for one alert, followed until it reaches something physical. Every arrow is a foreign key or a validity-interval lookup that Sec. 10.3 and Sec. 10.4 already built.

Read Figure 10.3 once for its depth rather than its detail. The alert is four levels from a sensor's calibration record, and each level is a place where the chain can be broken by a single unrecorded write.

Nothing in that tree is exotic. Every arrow is a foreign key or a validity-interval lookup that Sec. 10.3 and Sec. 10.4 already built. What makes provenance hard is not the mechanism; it is that **every arrow must be recorded at the moment it is traversed**, and each one is individually skippable. Twin work that takes traceability seriously makes

the same point from the other direction: data whose origin and method of computation stay fully traceable is what lets the whole of it be linked together meaningfully [6]. Infrastructure efforts for scientific twins put provenance tracking alongside data-quality validation as the pair that delivers transparency and reproducibility [7].

10.6.2 Chapter 7's five questions, answered against a schema

Chapter 7 Sec. 7.7.3 listed the five questions an incident review asks. This is the point of the chapter, so here they are with the field that answers each.

#	Chapter 7's question	Answered by
1	What did the twin predict, and what did it measure?	`twin_state` (prediction, at `as_of` and `decision_time`) and `measurement` (measured, at `event_time`) – **two tables, never one, per Chapter 5 Sec. 5.2.4**
2	Which model version and which parameter set produced that prediction?	`twin_state.model_version`, `twin_state.parameter_set_version`
3	When were those parameters last fitted, on what data window?	The parameter set's context row: `valid_from`, plus the dataset definition it was fitted on (Sec. 10.7)
4	Was the input inside the validity envelope at the time?	The envelope is a context row with a validity interval; the inputs are measurements at known `event_time`. One join
5	Was the credibility argument current, or had it expired?	The argument is a context row with `valid_from`, `valid_to` and its expiry conditions, and the expiry monitors write their own rows

All five are one join away, and none of them is answerable at all if the schema was designed for the present tense. That is the whole argument of this chapter compressed into a table, and it is worth noticing that four of the five are answered by *context with validity intervals* rather than by anything clever.

10.6.3 Chapter 8's three artifacts

Chapter 8 Sec. 8.2.3 required that a learned model's version identify three things – code, weights, and the data it was trained on – and that the training range be stored with the model. That is a context row:

```
model_version(
```

```
    model_id, version,
    code_hash, weights_hash, dataset_id,
    training_range,           -- per input dimension, Ch8 Sec. 8.6.2
    valid_from, valid_to,
    trained_at, trained_by
)
```

Two notes that make this work in practice.

`dataset_id` points at a definition, not a copy. Sec. 10.7 is what makes that safe. A copy of the training data is a second source of truth that will diverge; a definition that re-materialises the same rows is not.

The hashes are the cheap part and the load-bearing part. A content hash takes one line to compute and answers "is the thing running the thing we validated?" – a question that is otherwise answered by trust. Chapter 7 Sec. 7.3.4 made reproducibility a precondition for having the credibility conversation at all; this is that precondition, for learned models.

10.6.4 Retroactive invalidation, which is the uncomfortable one

Chapter 8 Sec. 8.6.3's third expiry trigger: the training data's certification as "normal" is called into question. It invalidates the argument **backwards** – when somebody discovers the pump was intermittently failing during the month you trained on, every alert and every silence since deployment is in question.

What the data layer owes here is the ability to answer "what else is affected?", and it is a graph traversal over the provenance chain:

1. The suspect window is `2026-02-01` to `2026-02-28`.
2. Which datasets include rows from that window? – from the dataset definitions of Sec. 10.7.
3. Which model versions were fitted on those datasets? – from `model_version.dataset_id`.
4. Which twin states were produced by those model versions? – from `twin_state.model_version`.
5. Which alerts and commands came from those states? – from derived output.

Five queries, and every one of them is impossible without a field somebody had to decide to add. The alternative – which is what usually happens – is a meeting in which people try to remember which models were trained when, and a decision to re-validate everything because nobody can scope it.

> **The design instruction that follows.** Build the *downward* chain (this alert came from that model came from that dataset) and you get the *upward* one (this bad data affects these alerts) for free, because it is the same edges read backwards. Build neither and the cost of one bad month is the whole system's credibility, which is what Chapter 7 said the argument was for.

And a note on scope. Provenance is also a security property – the integrity of the acquisition and dissemination layers propagates into everything downstream, which is why twin threat classifications organise themselves by layer [8]. Chapter 3 Sec. 3.3.3 and Chapter 13 own that; this chapter builds the record, which is a precondition for either.

10.7 The reproducible dataset

10.7.1 What Chapters 7 and 8 assumed

Chapter 7 said "six dry overnight windows", "week 2", "week 3". Chapter 8 said "the training set" and "two years of building management system data". Chapter 9 said degraded windows are excluded. **Every one of those phrases was treated as though it identified a specific set of rows, and none of them does.**

Chapter 7 Sec. 7.1.3 already complained about the symptom without naming the cure: a modelling expert who asks for two weeks of held-out readings and gets a spreadsheet with the times rounded to the hour cannot do their job, and will not always tell you why. The cure is that a dataset is an object in the system, not a phrase in a conversation.

10.7.2 A dataset is a query, a watermark, and an exclusion list

Three parts, and dropping any one of them breaks reproducibility in a different way.

```
dataset(
    dataset_id, name, purpose,        -- 'pot-3 calibration, week 1 nights'
    query_definition,                 -- bindings, quantities, time ranges, quality p
    ingest_watermark,                 -- the max ingest_time included
    exclusions,                       -- window ids with a reason each
    created_at, created_by,
    row_count, content_hash           -- what it actually produced, once
)
```

Why the query alone is not enough. Re-run "week 3, quality = good" next month and you may get *more rows* than you got today, because Chapter 9's backfill delivers late data with old `event_time`s. That is the same late-arrival mechanism as Sec. 10.5.3, and it means a dataset defined only by event-time range is not stable. **The `ingest_watermark` is what freezes it**: include only rows whose `ingest_time` is at or below the watermark, and the same query returns the same rows forever.

Why the exclusion list is separate from the query. Sec. 10.3.4's degraded windows could be a predicate, but the reasons for excluding a window are not all mechanical – "the greenhouse door was left open", "a technician was working in bay 2" – and those cannot be derived. An explicit list with a reason per entry is also the artifact Chapter 7's credibility argument should cite: **a dataset that excluded three days and says why is more credible than one that excluded nothing.**

Why the `row_count` and `content_hash` are recorded once. They are the check. Re-materialise the dataset, compare, and either you have the same data or you have learned something important. This is the same instinct as Chapter 7's golden trajectories, applied to data rather than to behaviour.

What this makes possible, and each item is something an earlier chapter needed and could not have:

- Chapter 7's hold-out is now a named object that can be shown to have been created before the fix it validated (Sec. 7.4.6's burned-hold-out trap becomes *checkable*, not merely a discipline).
- Chapter 8's retraining can state which dataset each version used.

- Sec. 10.6.4's retroactive invalidation has something to traverse.
- A colleague can reproduce a result without asking you what you meant.

10.7.3 Feature stores and labels, named

Chapter 8 Sec. 8.0 deferred "labelling infrastructure and feature stores" here. Briefly, because the durable half is Sec. 10.7.2.

A feature store is a system that computes derived inputs for models once, stores them, and serves the same values to training and to inference. The problem it solves is real and twin-relevant: a feature computed one way in a training notebook and another way in the production path is a defect that produces a model that works in testing and not in service.

What to take from the idea without buying the technology. The property that matters is *one definition, two consumers*. In a twin that is often a single well-named view over the measurement store, joined to context by time per Sec. 10.4.5, used by both the training job and the runtime path. If you have one model and four features, that is enough. The technology becomes worth it at a scale most twins do not reach.

Labels. Chapter 8 Sec. 8.5.1 showed that a detector's ceiling is set by what its training data called normal, and Chapter 8's fix was a period somebody certified as fault-free. **A certification is a context row**, with a validity interval, a `changed_by`, and a reason – exactly Sec. 10.4.3's mechanism. That is the whole labelling infrastructure a twin of this size needs, and it is the difference between "somebody said February was normal" and a fact the retroactive-invalidation query in Sec. 10.6.4 can traverse.

10.8 The four things that change under you

Schema evolution, for a system whose data outlives its code. Four changes recur, and each has a right answer that is cheap in advance.

A unit changes. A firmware update starts reporting tenths of a degree instead of degrees. Because Chapter 9 put `unit` in the tuple and Sec. 10.2.5 kept it, this is detectable rather than catastrophic – **provided nothing downstream assumes the unit is constant per series.** Store the unit per row, and have the daily job of Sec. 10.2.4 alert on a series whose unit changed.

A quantity's meaning changes. The same name now means something subtly different – "moisture" from a different sensor model with a different response. This is *not* a unit change and no column will catch it. The only defence is Sec. 10.4.3's context history plus the discipline that a sensor replacement is a recorded event, so that somebody investigating a residual step on 14 August finds the reason.

A new quantity appears. Cheap, if the schema is `(binding, quantity, ...)` rather than a column per quantity. This is the main reason for the narrow tuple shape, and it is worth defending in a review against the proposal to widen the table for convenience.

The model changes shape. A new parameter, a new input, a new output. Chapter 7 Sec. 7.4.6 earned `k_day` and turned two parameters into three; Chapter 8's correction

added three inputs. Because parameter sets and model versions are context rows with validity intervals, this is an insert. **A `parameters` table with a column per parameter is the version of this that requires a migration every time a modeller has an idea**, which is often, and which is how a data layer becomes the thing that slows the project down.

> **The general principle, and it is the last one in the chapter.** Store the *shape* that will not change – an observation is a binding, a quantity, a time, a value, a unit and a quality – and put everything that will change into rows rather than into columns. That is not a sophisticated idea. It is just the one that survives ten years of a physical twin being maintained by people who do not know you exist.

10.9 Worked example: the demonstrator's store, built

Chapter 3 Sec. 3.7.2 gave this component eleven lines. Here it is in full.

10.9.1 The engines

Two, per Sec. 10.2.4 and the dual-core pattern [1]:

- **A time-series store** for `measurement` and `window_quality`.
- **A relational store** for all context and all derived output.

Derived output goes in the relational store, not the time-series one, and the reason is Sec. 10.5.2: it has a compound temporal key and it joins to model versions and parameter sets. It looks like a time series and is not one.

10.9.2 The tables

Table	Store	Key	Rows per year
`measurement`	time series	`(binding, quantity, event_time)`	about 1.16 M
`window_quality`	time series	`(binding, quantity, window_start)`	about 8,000
`context_binding`	relational	`(binding, valid_from)`	tens
`context_asset`, `context_instrument`, `context_actuator`	relational	`(id, valid_from)`	tens
`parameter_set`	relational	`(binding, version, valid_from)`	tens
`model_version`	relational	`(model_id, version)`	tens
`change_event`	relational	`(id)`	tens

Table	Store	Key	Rows per year
`dataset`	relational	`(dataset_id)`	tens
`twin_state`	relational	`(binding, as_of, decision_time)`	about 630,000
`residual`	relational	`(binding, dose_event_id, decision_time)`	about 8,800
`alert`	relational	`(id)`	tens

Twelve pots, `Ts` = 10 minutes, two doses a day. Measurement rows: 22 series x 144 x 365 = 1,156,320. Twin state: 12 x 144 x 365 = 630,720. Residuals: 12 x 2 x 365 = 8,760.

10.9.3 The five decisions that are not obvious

1. `quality` is not nullable, and failed polls write rows. Sec. 10.3.3. This is the decision most likely to be argued about and it is the one that makes Sec. 10.3.4's completeness a query.

2. The series key is `(binding, quantity)` and contains no hardware. Sec. 10.3.2. Sensor identity comes from `context_binding` by time.

3. Context is append-only with validity intervals. Sec. 10.4.3. Nothing in this store is updated in place except closing a `valid_to`.

4. `twin_state` carries `decision_time` and `inputs_watermark`. Sec. 10.5.2. This is the field that made Sec. 10.5.3's review possible.

5. Every dataset used by Chapter 7 or Chapter 8 is a `dataset` row. Sec. 10.7.2. Including the six calibration nights, which Chapter 7 described in prose and this store describes as an object.

10.9.4 Retention

Content	Full resolution	Then	Never delete
`measurement`	2 years	hourly rollups, 10 years	–
`window_quality`	forever (8,000 rows/year)	–	yes
all context	forever	–	**yes – this is the smallest and most valuable data in the system**
`twin_state`	2 years	daily summary	–
`residual`, `alert`	forever	–	yes
`dataset` definitions	forever	–	yes

Under a gigabyte for a decade (Sec. 10.3.5), and the rollup rule carries the warning that a residual cannot be recomputed from hourly means.

10.9.5 What Chapter 7's credibility argument gains

Chapter 9 Sec. 9.9.5 contributed a paragraph and two admitted limitations. This chapter contributes a shorter and better one:

> Measurements, context and derived output are retained with validity intervals; the parameter set, model version, input watermark and decision time behind every twin state and every alert are recoverable for the retention period. Calibration and validation datasets are named objects with a frozen ingest watermark and a documented exclusion list. **Limitation: changes to the physical twin enter the record only when a human enters them; an unrecorded repotting is invisible to every check in this document.**

That limitation is Chapter 7's expiry-trigger problem restated honestly, and it is the one thing in this chapter that infrastructure cannot fix.

10.10 Faded example: the turbine's data layer

Chapters 4 through 9 each took the offshore turbine one step further. Now store it. Two parts are worked; four are yours.

The system, recapped. A floating offshore wind turbine: a reduced-order structural model, a Kalman filter, a fatigue service reporting remaining life, Chapter 8's anomaly detector on residuals, and Chapter 9's three sampling rates – 2 kHz vibration, 1 Hz operational state, 10-minute environment – with the connector's boundary at the edge.

(a) Three rates means three retention policies – worked. Chapter 9 established that the high-rate streams do not leave the turbine intact. So the data layer has three tiers and they are not variations of one policy:

Stream	Kept at the edge	Shipped	Kept centrally
Vibration, 2 kHz	A rolling buffer of hours	Features per window, plus raw windows around triggered events	Features forever; raw event windows for the certification period
Operational, 1 Hz	–	All of it	Full resolution for years; it is small
Environment, 10 min	–	All of it	Forever; it is tiny and it is the context for everything else

And the consequence Sec. 10.3.5 warned about, in its strongest form. A feature is a decision about which questions can be asked later. The turbine's raw vibration is discarded within hours, so **any analysis nobody anticipated is permanently impossible** – not expensive, impossible. That is a decision worth taking to the fatigue engineers before it is taken by a default retention setting.

(b) The certification output changes the retention question – worked, because it is the answer people get wrong. Chapter 7 Sec. 7.9 established that this twin has

two outputs at two levels of rigour: maintenance scheduling (advisory) and life extension (certification, with a regulator). For the second, retention is not an engineering decision at all. The evidence supporting a life-extension claim must survive as long as the claim does, which is the remaining life of the asset – **so a 2026 decision to downsample can invalidate a 2041 certification.** Nothing in the data layer signals that. Somebody has to know to ask.

Now yours.

(c) Build the context model. Name at least six context entities for a floating turbine and, for each, one change that must be a `change_event` per Sec. 10.4.4. *Hint: Chapter 7's exercise 7.9(f) asked what happens after a mooring line is replaced; this is where that answer lives.*

(d) Sec. 10.4.5's time-join used `w` = 15 minutes for greenhouse air temperature. Work out `w` for joining a 1 Hz pitch-angle signal to a 2 kHz vibration channel, and say what Chapter 9 Sec. 9.5.3's clock requirement has to be for your answer to hold.

(e) Apply Sec. 10.5.2 to the fatigue service. It accumulates damage over years, so a single `as_of` and `decision_time` per output may not be the right shape. Decide what its derived-output schema should be, and what "what did the twin believe at time t" means for a quantity that is an integral of everything before t.

(f) Sec. 10.6.4's retroactive invalidation is a five-step traversal. Run it for this asset given the discovery that one accelerometer was mis-mounted for three months. Say which of the five steps is hardest here and why – and whether the answer would have been different had the raw vibration data been retained.

10.11 Posed problem: the data-layer review

No solution is given.

The situation. A utility has been running a twin of 400 district-heating substations for two years – the system whose ingest you designed in Chapter 9 Sec. 9.11. It detects failing heat exchangers from a persistent change in the difference between supply and return temperature.

Last month a regulator asked for evidence supporting three alerts raised in the previous year. The team could not produce it. In the post-mortem, these facts emerged:

- Measurements are stored as `(substation_id, timestamp, supply_temp, return_temp, flow)` – one wide row per reading, no quality column, nulls where a value was missing.
- There is a `substation` table with the current sensor model, current calibration date, and current heat-exchanger type. It is updated in place by a maintenance web form.
- Alerts are stored with a timestamp and a message string.
- The model is retrained quarterly on "the last two years of data" and the fitted parameters are written to a config file in version control.
- Nobody can say which substations had sensors replaced during the year in question, because the maintenance form overwrote the previous values.

Produce a review of no more than four pages containing:

1. **A defect list**, ordered by which are impossible to fix retroactively versus merely expensive. Say explicitly which of the regulator's questions can never now be answered for the period in question.
2. **The measurement schema you would replace theirs with**, per Sec. 10.3.3, and what the wide-row shape costs them beyond quality (*hint: Sec. 10.8's third change*).
3. **The context model**, per Sec. 10.4.3, including how you would migrate a two-year-old update-in-place table into validity intervals, and what you would honestly have to mark as unknown.
4. **What derived output must be retained** and the two timestamps it needs, per Sec. 10.5.2, with one worked example of a question that becomes answerable.
5. **A dataset definition** per Sec. 10.7.2 for the quarterly retraining, and a statement of what "the last two years of data" currently fails to specify.
6. **The retroactive-invalidation traversal** they would need if one substation's sensors turn out to have been mis-calibrated for six months, and which link in the chain they are missing.
7. **A retention policy** with the correctness argument, per Sec. 10.3.5, including how long the regulator's interest implies.
8. **One paragraph on what this costs and what it does not buy.** Be honest: the fix does not make the twin more accurate, and somebody will ask why they should pay for it.

What a good review looks like. It separates "we cannot answer this now" from "we cannot answer this ever", because the second list is the one that justifies the work. It notices that a fault detector on a *difference* between two sensors makes sensor-replacement history load-bearing, not optional. It resists rewriting the whole system, and instead identifies the two or three changes that stop the bleeding – add a quality column, stop updating context in place, and name the datasets – while accepting that the past two years are partly unrecoverable and saying so plainly. And its last paragraph is honest that the value is entirely in questions that have not been asked yet, which is a genuinely hard case to make and is made better by admitting it.

10.12 Summary

Seven things, tied to the five objectives.

1. **A value without its context is not data, it is a number** (Sec. 10.1). Chapter 3 attributed that sentence to this chapter seven chapters early. The data layer is underestimated because the measurement part is the tractable quarter, the context is small enough to feel unimportant, and the parts that matter are exactly the parts that cannot be added retroactively.
2. **Three kinds of content, three shapes, one binding, not one engine** (Sec. 10.2). And two column names pinned, because Chapter 3 gave each under two spellings: `quantity` and `event_time`. *(Objective 1.)*
3. **The measurement schema either represents absence or destroys it** (Sec. 10.3.3). `quality` not nullable, rows written for failed polls, and `substituted`

distinguishable from measured – because Chapter 9's exclusion rules are `WHERE` clauses and a `WHERE` clause needs a column. The series key holds what identifies the measured thing, never the measuring apparatus. *(Objective 2.)*

4. **Context has a history and that is the whole difficulty** (Sec. 10.4). Validity intervals, never update in place, `valid_from` at the *physical* time, and an enumerated set of change types. A repotting on 14 August is then one row, and every query about August's readings gets August's sensor. *(Objective 3.)*
5. **Derived output needs a timestamp measurements do not have** (Sec. 10.5). `as_of` is the moment a belief is about; `decision_time` is the moment it was formed; `inputs_watermark` is what the twin was allowed to know. On 14 August the observed step was 57.0 and the twin believed it was 0, and **both are correct** – one is about the world, one about the system. Without those fields the November reviewer concludes the alert was a bug and hunts a defect that is not there. *(Objective 4.)*
6. **Provenance is a chain of ordinary joins that must each be recorded at the moment they are traversed** (Sec. 10.6). Chapter 7's five incident questions are each one join away, and four of the five are answered by context with validity intervals rather than by anything clever. Build the downward chain and you get retroactive invalidation for free, because it is the same edges read backwards. *(Objective 5.)*
7. **A dataset is a query, a watermark and an exclusion list** (Sec. 10.7). Chapters 7 and 8 said "week 3" and "the training set" as though those identified rows. The watermark is what makes a dataset stable against Chapter 9's backfill, and the exclusion list with a reason per entry is what makes it credible. *(Objective 5.)*

And the note this chapter adds to Part III. Chapter 9 found that a large part of whether a twin deserves to be believed is settled before any model runs. Chapter 10's version is narrower and sharper: **most of what an incident review will need is decided by schema choices made before anybody knows there will be an incident**, and every one of those choices is cheap today and unavailable later. That asymmetry is the whole argument for taking this chapter seriously, and it is why the data layer is the part of a twin that is most worth over-building slightly.

10.13 Exercises

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

10.13.1 *(Objective 1.)* Classify each as measurement, context or derived output, and say which store it belongs in: (a) the pump's calibration slope; (b) a moisture reading; (c) the twin's moisture estimate; (d) the fact that a poll failed at 14:20; (e) the alert raised at 17:22; (f) yesterday's completeness figure for pot 3.

10.13.2 *(Objective 2.)* A colleague proposes the series key `(site, pot, quantity, sensor_model)` because "we'll want to compare sensor models". Give the objection, and say how the comparison they want is done under Sec. 10.3.2's rule.

10.13.3 *(Objective 2.)* Write the predicate for "genuine measurements only" over Sec. 10.3.3's schema, then write the predicate for "everything the connector knows about this window, including its failures". Say which one Chapter 7's calibration uses and which one Sec. 10.3.4's completeness uses, and why they differ.

10.13.4 *(Objective 3.)* Pot 7's sensor is replaced on 3 September at 14:30. The technician records it on 5 September at 09:00, entering the correct time of the work. Write the two rows of `context_binding` before and after, and say what would have gone wrong had `valid_from` been set to the recording time instead.

10.13.5 *(Objective 3.)* You must join a pot's moisture to greenhouse humidity for Chapter 8's learned correction. The nearest humidity sample to a given moisture sample is 6 minutes away; for one moisture sample it is 95 minutes away because of an outage. State what your join does in each case, what quality the joined row carries, and whether that row may be used for training.

10.13.6 *(Objective 4.)* For each, say whether it needs `decision_time`: (a) a dashboard showing current moisture; (b) the estimate that a command was based on; (c) a nightly report of yesterday's mean temperature; (d) a forecast published to an operator; (e) a raw sensor reading.

10.13.7 *(Objective 4.)* Rework Sec. 10.5.3 with one change: the 17:10 reading arrives at 17:19, before the service evaluates at 17:22. Give the answers to all four queries now, and say what this shows about the relationship between Chapter 9's watermark and Chapter 11's evaluation schedule.

10.13.8 *(Objective 5.)* Chapter 7's hold-out was "week 3". Write it as a Sec. 10.7.2 dataset definition, inventing plausible values, and name the one field that would have made Chapter 7 Sec. 7.4.6's burned-hold-out trap detectable rather than merely a discipline.

10.13.9 *(Objective 5.)* Six months after deployment, a sensor is found to have been reading 4 per cent low since installation. Run Sec. 10.6.4's five-step traversal for the demonstrator and list what you would notify. Then say which single missing field would make the traversal impossible.

10.13.10 *(Objectives 1-5, and the one that leaves the book.)* Take the week of real data from exercise 9.13.10. Load it into a store built to Sec. 10.9's schema. Then answer, from the store alone and without asking anyone: which sensor produced each reading, how complete each day was, and which windows you would exclude from a calibration. Report which of the three you could not answer and what field was missing.

Solutions and hints

10.13.1. (a) Context, relational – and it has a validity interval, because pumps get recalibrated. (b) Measurement, time series. (c) Derived, relational, with `as_of` and `decision_time`. (d) **Measurement**, time series – a row with a null value and `quality = no_reading`. This is the one people get wrong; a failed poll is a recorded fact about the world, not an absence. (e) Derived, relational. (f) Derived in the general sense, but it belongs with the measurements as `window_quality` (Sec. 10.3.4), because it is about the stream rather than about the physical twin.

10.13.2. Objection: `sensor_model` is a property of the measuring apparatus, not of the measured thing, and it changes – so replacing a sensor splits pot 7's history into two series, and every range query, alert and calibration must thereafter union an unknown number of them. The comparison they want is done by joining measurements to `context_binding` by time (Sec. 10.4.3) and grouping by the sensor model that was valid then – which is

more capable than putting it in the key, because it can also answer "did the residuals change at the swap?", a question the split-series design makes awkward.

10.13.3. Genuine measurements: `quality = 'good'`. (Chapter 9's table excludes `uncertain` from calibration too, so a stricter reading is `quality = 'good'` exactly – and if you wrote `value is not null`, that is the bug this exercise is for, because it admits `substituted`, `out_of_range` and `uncertain`.) Everything the connector knows: no predicate at all – every row, including the null-valued ones. Calibration uses the first; completeness uses the second as its denominator input and the first for `good_count`. They differ because completeness is a question about the *stream* and calibration is a question about the *world*.

10.13.4. Before: `(pot-7/moisture, S-0210, valid_from 2026-06-01 09:00, valid_to null)`. After the entry on 5 September: the first row's `valid_to` becomes `2026-09-03 14:30`, and a new row `(pot-7/moisture, S-0318, valid_from 2026-09-03 14:30, valid_to null)` is inserted. Had `valid_from` been 5 September 09:00, then **42.5 hours of readings from the new sensor would be attributed to the old one** – and worse, they would be attributed to a sensor whose calibration record is the wrong one, so any residual anomaly in that window would be investigated against the wrong instrument. Note that the row is *inserted* two days late; lateness of insertion is fine, wrongness of `valid_from` is not.

10.13.5. At 6 minutes: within any sensible `w` for greenhouse humidity (Sec. 10.4.5 suggested 15 minutes), so join it, and mark the joined humidity column as matched-not-measured. At 95 minutes: **refuse**. The bounded nearest join returns nothing, and the row either carries a null humidity with an appropriate quality or is dropped from the feature set. May it be used for training? The 6-minute row, yes, with its provenance recorded. The 95-minute row, no – and note that this is Chapter 8 Sec. 8.5.1's lesson arriving as a join predicate: a training set that quietly substitutes a 90-minute-old humidity has taught the model something false about the relationship it is being fitted on.

10.13.6. (a) No – current state, nobody will litigate it. (b) **Yes, absolutely** – this is the case Chapter 2 Sec. 2.8 and Chapter 7 Sec. 7.7.3 exist for. (c) No, if it is a report of measurements; yes, if it is a report of *estimates*, because those are beliefs. (d) Yes – a forecast an operator acted on is a decision input, and "what did we tell them, and when" is the first question after anything goes wrong. (e) No – a measurement is not a belief. It has `event_time` and `ingest_time` and that is the right pair.

10.13.7. Query 1: 663, unchanged. Query 2: 57.0, unchanged. Query 3: the twin now believed the step was 57.0, because its `inputs_watermark` at 17:22 was 17:10. Query 4: no alert was raised, correctly. **What it shows:** the incident of Sec. 10.5.3 was caused by nine minutes of arrival delay interacting with an evaluation schedule that did not respect Chapter 9's lateness allowance – and the same code, same data and same thresholds produce the right answer when the timing is nine minutes different. That is the signature of a race condition. Note also what stays hidden in this version: the service still subtracts without checking that its post-dose operand is actually after the dose, and here it gets away with it because the operand happens to be a genuine post-dose reading. **The second defect of Sec. 10.5.3 is latent under good timing**, which is why the fix needs both parts – the operand check and the evaluation schedule – rather than whichever one the incident happened to expose.

10.13.8. *Hint plus the answer to the last part.* A plausible definition: `name` "pot-3 hold-out week 3"; `purpose` "validation of three-parameter model"; `query_definition` binding `pot-3/moisture`, `event_time` in `[2026-07-20, 2026-07-27)`, `quality = 'good'`; `exclusions` none; `created_at` a date; `row_count` and `content_hash` as produced. **The field that makes the burned-hold-out trap detectable is `created_at`** – compare the dataset's creation time against the `trained_at` of the model version it validated, and a hold-out created *after* the fix it is supposed to validate is a query, not a matter of professional discipline. (`ingest_watermark` is what makes it *reproducible*; `created_at` is what makes the ordering *checkable*. Both matter and they do different jobs.)

10.13.9. The traversal: (1) suspect window is installation to discovery, for that one binding; (2) which datasets include rows from that binding and window – the calibration campaigns and any hold-out; (3) which parameter sets and model versions were fitted on them; (4) which `twin_state` rows were produced by those versions; (5) which alerts and, if the loop is closed, which commands came from those states. Notify: whoever consumed the alerts, and whoever relies on the credibility argument, which has now expired retroactively per Chapter 8 Sec. 8.6.3. **The single missing field that would make it impossible: `twin_state.parameter_set_version`.** Without it, step 4 cannot be done at all, and the only honest scope is "everything since installation" – which is the outcome Sec. 10.6.4 said the traversal exists to avoid.

10.13.10. No solution. One prediction: you will be able to answer the completeness question, you will be able to answer the sensor question only if you wrote down the sensor identity at the start of the week, and the exclusion question will turn out to depend on something that happened in the room that nobody recorded. That third answer is the chapter's thesis, and it is more convincing when it happens to you than when it is asserted here.

10.14 Where to go next

In this book. Chapter 11 is the immediate consumer: every service it builds reads from these tables, and the fix identified in Sec. 10.5.3 – evaluate a dose only when its window is complete or the lateness allowance has expired – is Chapter 11's to implement. Chapter 12 selects the engines this chapter deliberately declined to name, and is where the build-versus-buy question for a twin platform's data layer is answered; open-source twin frameworks differ substantially in what they provide here [9], [10]. Chapter 13 covers the standards, which for this chapter means the asset-model and information-model work that Sec. 10.4.6 named without specifying [11], [12]. **Chapter 14 is where this chapter's tables get operated**: migrations, backfills, the day the retention policy is exercised for the first time, and the human process that Sec. 10.9.5 admits is the weakest link. Chapter 15 asks what happens when 400 substations become 400 twins, and the context model is the part that does not scale by copying.

In the literature, if you want more.

- *The storage split:* [1] proposes the relational plus time-series dual-core architecture that Sec. 10.2.4 and Sec. 10.9.1 follow, and is the clearest statement of why one engine is the wrong answer.

- *Context as a first-class, changing thing:* [4] is the greenhouse exemplar this book has cited since Chapter 1, and it is built around recording how the physical system is structured *and how that structure changes* – with plants that move, which is Sec. 10.4's problem in its purest form. [13] takes the same line further into semantic technology than this chapter goes.
- *Provenance:* [6] on keeping origin and method of computation traceable throughout, and [7] on provenance tracking paired with data-quality validation as the basis for reproducibility.
- *Change over time:* [5] distinguishes continuous change such as wear from discontinuous change such as replacement, which is the distinction Sec. 10.4.4's enumerated change types encode.
- *When volume forces the policy:* [2] and [3], as in Chapter 9 – Sec. 10.10(a)'s three-tier retention is what those architectures look like from the storage side.
- *Consulted, not drawn on above:* [8] for the layered view of why provenance is also a security property, [14] on ontology-driven service engineering with a knowledge graph for model management, and [15] on the platform properties whose measurement this store makes possible.

In the demonstrator. Exercise 10.13.10 is the assignment and it needs no model, no simulator and no machine learning – only Chapter 9's week of real readings and the schema of Sec. 10.9. Load it, then try to answer the three questions from the store alone. The one you cannot answer is the one this chapter was written about, and finding out which it is takes an hour.

10.15 References

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

[2] L. Gigli *et al.*, "Next Generation Edge-Cloud Continuum Architecture for Structural Health Monitoring," *IEEE Transactions on Industrial Informatics*, vol. 20, no. 4, pp. 5874–5887, Apr. 2024, doi: 10.1109/TII.2023.3337391.

[3] F. Zonzini *et al.*, "Structural Health Monitoring and Prognostic of Industrial Plants and Civil Structures: A Sensor to Cloud Architecture," *IEEE Instrumentation & Measurement Magazine*, vol. 23, no. 9, pp. 21–27, Dec. 2020, doi: 10.1109/MIM.2020.9289069.

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

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

[6] M. Heithoff, M. Trinh, J. Michael, B. Rumpe, and C. Brecher, "A Digital Shadow for Accurate Robot Motion Control: Integrating Data with Friction Models," Grand Rapids, USA, Jul. 2025. doi: 10.5283/epub.77667.

[7] "interTwin: Advancing Scientific Digital Twins through AI, Federated Computing and Data."

[8] C. Alcaraz and J. Lopez, "Digital Twin: A Comprehensive Survey of Security Threats," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 3, pp. 1475–1503, 2022, doi: 10.1109/COMST.2022.3171465.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] E. Kamburjan, N. Bencomo, S. L. Tapia Tarifa, and E. B. Johnsen, "Declarative Lifecycle Management in Digital Twins," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, Linz Austria: ACM, Sep. 2024, pp. 353–363. doi: 10.1145/3652620.3688248.

[14] B. Oakes, C. Gomes, E. Kamburjan, G. Abbiati, E. Ecem Bas, and S. Engelsgaard, "Towards Ontological Service-Driven Engineering of Digital Twins," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, in MODELS Companion '24. New York, NY, USA: Association for Computing Machinery, Oct. 2024, pp. 464–469. doi: 10.1145/3652620.3688261.

[15] 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.