Chapter 9 – Connecting the Physical: Sensors, Protocols, and Streaming Data

9.0 Before you start

Where we are. Part II is over. You spent five chapters learning enough about models, simulation and credibility to hold up your end of a conversation with people who build them. Part III builds the system those models live in, and it is the part of the book where the work is yours.

Chapter 3 drew seven components and this chapter builds the first one. Four separate chapters have now written the sentence "Part III builds: Chapter 9 the connector", and Chapter 3 Sec. 3.2.1 said in as many words that Chapter 9 is where the protocols themselves live.

The register changes here, and it is worth naming the change. Part II was written against a reader who could not evaluate a modelling claim, so everything was explained. That is the wrong posture now. You have written HTTP clients. You have consumed message queues. You have lost an evening to a duplicate delivery and an afternoon to a timezone. Explaining what a broker is would be the insult that Part II's explanations were guarding against in the other direction.

So:

> **This chapter does not teach protocols and does not compare them. It teaches what a *twin* needs from a connection, which is a different and much shorter list than what any protocol offers.**

You can read a specification without help. What no specification tells you is which of its features your twin's credibility argument depends on, and that is the whole subject here.

What you are assumed to know. Everything so far, plus ordinary distributed-systems experience. Specifically from this book: Chapter 1's value metric and its Sec. 1.8.1 description of the demonstrator hardware; Chapter 2's twinning rate, age of twin and time discrepancy; Chapter 3's seven components, the connector's tuple, the binding, and the derivation of `Ts` = 10 minutes; Chapter 4's assumption ledger, especially A2 and A3; Chapter 5's distinction between `h` and `Ts`; Chapter 7's calibration windows, hold-out and credibility argument; Chapter 8's input-coverage check.

The maths budget, loosened in exactly one direction. More arithmetic is now welcome; no new mathematics is. Still no derivatives, integrals, matrices or probability. There are four arithmetic set pieces – a data-volume calculation, a clock-drift calculation, a completeness count, and a stuck-sensor threshold – and all four are multiplication and division. **The Nyquist-Shannon sampling theorem is deliberately not stated here**; Sec. 9.3.2 teaches aliasing as a named phenomenon with one concrete instance and points at where to learn it properly.

One warning about numbers, carried forward from Chapters 4, 7 and 8. Where this chapter puts figures on the demonstrator's sensors, those figures are illustrative. The hardware is real and documented; no measurement campaign has been run on it for this book, and none is invented here. The *method* transfers; check the datasheet you actually have.

What this chapter deliberately does not cover. Protocol specifications and protocol comparison – named for recognition, then Chapter 12 for tool selection and Chapter 13 for standards. Storage – Chapter 10, and this chapter is careful to say what the connector must *record* without saying where it lives. Signal processing, compression and sensor fusion – named with pointers, not taught. Networking below the transport. The command path – Chapter 3 Sec. 3.2.7 owns it, and Chapter 3's rule that the two directions do not share a box holds here. Security in general – Chapter 3 Sec. 3.3.3 and Chapter 13. Deployment topology – Chapter 12.

Learning objectives

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

1. **Read** a sensor datasheet for the six numbers that change a twin's design, and **decide** whether a proposed measurement can support a stated decision.
2. **Derive** a sampling interval from a decision, **recognise** aliasing as a failure a correctly derived interval can still have, and **design** the empirical check for it.
3. **Choose** between polling and subscription, and **evaluate** any transport against the five properties a twin needs from one.
4. **Distinguish** the four timestamps in an ingest path, **compute** Chapter 2's twinning rate and time discrepancy from them, and **size** an acceptable clock error against the sampling interval.
5. **Detect** the four ways a measurement stream goes wrong and **specify** the quality states the rest of the twin consumes.

9.1 The connector, and why its cost is always underestimated

Chapter 3 gave the connector one job and one output contract. Both are worth restating, because everything in this chapter is an elaboration of them.

The job. Speak the physical twin's protocol and present it to the rest of the system as something ordinary.

The output. A stream of tuples:

```
(binding, quantity, value, unit, timestamp, quality)
```

plus an honest answer to "are you currently connected?"

That is a small contract, and the smallness is the design. Everything upstream – store, estimator, runner, services – sees one normalised form, and the diversity of the physical world stops at one component. The recurring shape in the literature is the same: isolate the protocol behind per-protocol handlers, or put a broker in the middle so that twins and devices need not know each other's transport [1]. Industrial architecture guidance treats connectivity as a framework in its own right with a named middleware set [2].

Now the part nobody budgets for. In project after project, the connector is estimated as "read the API" and delivered as a quarter of the engineering. Three reasons, and none of them are protocol difficulty.

The physical twin was not designed for you. Its interface exists for whatever the people who built it needed. It may have no way to ask for history, no way to say when it last succeeded, and no notion of your binding. You architect against the interface that exists, which Chapter 3 Sec. 3.7.1 already flagged and which is worth internalising as a rule rather than a complaint.

Every failure in the physical world arrives through this component. A loose connector, a flat battery, a technician unplugging something, a firmware update that changes a unit – all of it becomes, at your boundary, some tuple or some absence of one. Sec. 9.7 is entirely about that.

The connector is where the twin's credibility begins. Chapter 7 built a credibility argument on a hold-out week. That week is a set of tuples this component produced. If it silently forward-filled a gap, Chapter 7's spread of 2.0 is a fiction and nothing downstream can tell. **The most consequential code in a twin is often the least interesting-looking code in the twin.**

9.2 Sensors: what you are actually given

9.2.1 A sensor is a model with a datasheet

The instinct is to treat a sensor as a source of truth with some noise on it. It is more useful to treat it as **a model of the world that somebody else built and documented**, with all of Chapter 4's apparatus applying.

A capacitive soil-moisture sensor does not measure water. It measures a capacitance, converts it to a number, and somebody has claimed a relationship between that number and water content. Chapter 4's assumption A2 is exactly that claim – the reading is affine in water content – and Chapter 4 already recorded what happens when it breaks: near saturation the sensor flattens, a real change produces no reading change, and the twin sees a fault that is not there.

So the sensor has an assumption ledger too, and its rows are in the datasheet. The reference treatment for twins makes the same point and takes a thermistor as its running example: the relationship between the physical quantity maps to the reported value can be established by experiment, and the datasheet usually states it [3].

9.2.2 Six numbers that change your design

Datasheets are long. Six numbers do the work.

Number	What it means	What it changes
Range	The span the sensor is specified over	Chapter 7's validity envelope, and Chapter 8's input-coverage check. Outside it, the reading is not wrong – it is undefined

Number	What it means	What it changes
Resolution	The smallest change reportable	Whether a change your model predicts is even visible. Sec. 9.7.4 derives a threshold from this
Accuracy	How far a reading may be from the truth	Chapter 7's residual spread has a floor here. No model beats its instrument
Drift	How fast accuracy degrades	Chapter 7's expiry conditions. A sensor specified at 1 per cent per year is a recalibration schedule
Response time	How long the reading takes to reflect a change	An upper bound on useful sampling rate, and the reason a fast poll can return stale physics
Maximum sample rate	How often it can be read	The ceiling on `Ts`, and sometimes a power budget

Resolution and accuracy are not the same number and confusing them is the commonest datasheet error. A sensor reporting to 0.01 units with an accuracy of 2 units gives you four decimal digits of which two are decoration. You will see the extra digits in the store, in the dashboard, and eventually in a residual plot where somebody reads meaning into them.

Two of the six reach into Part II directly. Accuracy sets a floor under Chapter 7's residual spread: a model validated to plus or minus 2.0 reading units against a sensor accurate to plus or minus 3 has been validated against noise. And resolution decides whether an effect exists for you at all – if the model predicts a 0.7-unit change and the sensor reports whole numbers, that change is invisible in a single sample no matter how good the model is.

9.2.3 The demonstrator's sensors, read that way

Chapter 1 Sec. 1.8.1 listed the hardware from the demonstrator's own documentation. Read as a connector designer rather than as a shopping list:

Sensor	Quantity	Bus	What the connector must know
Adafruit STEMMA soil moisture, one per plant	Soil moisture, as a raw reading	I2C via a TCA9548A multiplexer	Several sensors share an address, so the multiplexer channel is part of the binding. Get that wrong and you attribute pot 3's water to pot 7
SHT45	Air temperature and relative humidity	I2C	One sensor for the whole greenhouse, so its binding is the greenhouse, not a pot. Chapter 8's learned correction consumes both
AS7341	Ambient light, several spectral channels	I2C	One reading is several quantities. The tuple is per quantity, so one poll produces many tuples
MODBUS soil probe (**optional**)	Soil temperature, electrical conductivity, moisture	MODBUS	Different bus, different failure modes, and a second moisture measurement that will disagree with the first
CS-IO404 relay / AD20P pump	Watering events	MODBUS	Not a sensor, but the source of a stream the twin needs: what was actually commanded, and when

Four things in that table are the chapter in miniature.

The multiplexer channel is part of the binding. Chapter 3 Sec. 3.3.1 made identity and binding a cross-cutting concern; here is where it becomes a wiring diagram. An identity defect in a connector does not look like a bug. It looks like pot 3 drying strangely.

One device can produce many quantities. The AS7341's channels, and the SHT45's two. A connector modelled as one-reading-per-poll will fight this forever.

A second measurement of the same quantity is a design decision, not a bonus. The optional

MODBUS probe also reports moisture. Two moisture numbers for one pot that disagree is not redundancy unless somebody decides in advance which is authoritative, or how they combine.

Watering events are ingest, not actuation. The twin needs to know what the pump actually did. That stream arrives through the connector's ingest path even though it concerns the actuation side, and Chapter 3's insistence that the two directions are different boxes survives intact: reading the watering history is not commanding a pump.

9.2.4 Instrumentation is a cost with a decision attached

Chapter 1 Sec. 1.7 listed four situations in which a twin is the wrong answer, and the fourth was: the measurement does not exist and cannot be added – no sensor, no twin – and **the instrumentation cost belongs in the estimate, not in a later phase**. This is the later phase, and here is what it looks like when the bill arrives.

The MODBUS soil probe is *optional* in the demonstrator's hardware list. Should the twin use it? Chapter 1's method answers this, and it is worth running because the shape of the answer generalises to every "should we add a sensor" question you will be asked.

Step 1 – what would it let you decide that you cannot decide now? It adds soil temperature and electrical conductivity, and a second moisture reading. Chapter 1's value metric is experiment-weeks lost to undetected watering faults. Soil temperature does not bear on that. Conductivity does not either.

Step 2 – what would it let you decide better? The second moisture reading is a genuine candidate. Chapter 4's assumption A3 said a dose that misses the sensed soil is indistinguishable from a pump failure; a second probe elsewhere in the pot would sometimes separate them. And Chapter 7's credibility argument admits that limitation in writing.

Step 3 – what does it cost, in full? The probe, a MODBUS interface, a second decoder in the connector, a second binding, a second calibration campaign (Chapter 7 needed two experiments for two parameters, and this adds a third quantity), a second set of failure modes, and a rule for what to do when the two moisture readings disagree. **That last one is the expensive part and it is always omitted from the estimate.**

Step 4 – compare with the alternative. Chapter 8 Sec. 8.7.3 already looked at a camera for the same limitation, and a camera separates the two faults cleanly rather than sometimes.

The decision, and the reasoning that transfers. Not for the fault detector. A second sensor that *sometimes* separates two faults, both of which send the same technician to the same rig, does not change what anybody does – and Chapter 1's rule is that fidelity not demanded by the value metric is unpaid work. It might well be worth it for a different twin of the same pot, one scheduling doses to a target moisture, where soil temperature genuinely bears on the decision.

> **The general form.** A sensor is worth adding when it changes a decision, not when it adds a quantity. And the cost of a sensor is not the sensor: it is the decoder, the binding, the calibration, the failure modes and the disagreement policy.

9.2.5 What to ask

- What are the range, resolution, accuracy, drift, response time and maximum sample rate, and which of them is closest to biting?
- What does this sensor actually measure, and what is the claimed relationship to the quantity we care about?
- What does it report when it is unhappy, and is that value distinguishable from a real reading?
- Who calibrates it, how often, and how will we know it has drifted?

9.3 Sampling: how often, and what you lose

9.3.1 `Ts` comes from the decision

This is settled and only needs assembling. Chapter 2 Sec. 2.7.1 said "real time" is not a requirement but an abdication, and gave the useful question: how stale can the digital state be before the decision it feeds becomes wrong? Chapter 3 Sec. 3.7.1 answered it for the demonstrator – doses are twice daily, the moisture step resolves over tens of minutes, so `Ts` = 10 minutes – and Chapter 5 Sec. 5.4.5 insisted that `Ts` is not the solver's step size `h`, and that `Ts` belongs to the connector.

It belongs to you. Here is the derivation as a procedure, since Chapter 3 gave it as a conclusion:

1. **Name the decision and its latency requirement.** "Tell a technician a dose did not land, within an hour."
2. **Name the physical event the decision depends on** and how long it takes to become visible. The moisture step from a dose resolves over tens of minutes.
3. **Sample fast enough to see the event, several times.** Three to five samples across the event is a working convention – enough to see a step as a step rather than as a single jump.
4. **Check the ceiling.** Sensor maximum rate, bus contention, power, data volume (Sec. 9.8.1).
5. **Check the floor.** The latency requirement from step 1. A ten-minute interval cannot support a one-minute alerting requirement, whatever else is true.

Ten minutes satisfies all five for the demonstrator, and the derivation took no theory.

9.3.2 What a correctly derived interval can still miss

Here is the failure that step 3 above does not prevent, and it is the one worth knowing by name.

Aliasing is fast behaviour appearing as slow behaviour because the sampling was too slow to see it. Not "you missed some detail" – something qualitatively different: the samples describe a pattern that *does not exist*.

A concrete instance in the demonstrator. Suppose the greenhouse's ventilation fan runs on a **10-minute** cycle – five on, five off – and dries the pots slightly while running. You sample moisture every 10 minutes.

Case 1: the periods match exactly. Every sample lands at the same point in the fan's cycle – always mid-blow, or always mid-rest, depending only on when you started. What you record is a **flat, clean level**, and it is not the pot's average moisture. It is one phase of an oscillation, mistaken for a steady state. Nothing in the data hints that the fan exists.

Case 2: the periods nearly match, which is the realistic one. Suppose the fan's cycle is 10 minutes and 6 seconds. Now the sampling point walks slowly through the fan's cycle, and the recorded moisture rises and falls smoothly with a period of

`(600 x 606) / (606 - 600) = 363,600 / 6 = 60,600 seconds = about 17 hours`

A 10-minute fan has produced an apparent 17-hour cycle in your data. It is smooth, it is strong, it looks exactly like a daily environmental rhythm, and it does not exist. Worse: Chapter 4's assumption A1b is *about* a day-night difference in drying rate, so Chapter 7's residuals will develop a bias, Chapter 7 Sec. 7.4.6 will attribute it to that row, an extra parameter will be earned on data that appeared to justify it, and it will fix nothing – because the effect is in the connector.

That is what makes aliasing different from ordinary measurement error. A noisy reading is visibly noisy. An aliased one is clean, plausible, consistent, and wrong, and every diagnostic technique in Part II is designed to find structure in data rather than to doubt that the structure is real.

Why the chapter stops there. There is a theorem about how fast you must sample to avoid this, and it is properly presented – with the conditions it needs, and what quantisation does to them – in the twin sensing literature [3]. Read that when you need it. What you need in a design review is the name, the recognition that a decision-derived interval does not rule it out, and the check below.

9.3.3 The empirical check, which costs an afternoon

Sample much faster than `Ts`, once, for a short window, and compare.

Run the sensor at its maximum rate for an hour. Then take every Nth sample to reconstruct what your chosen `Ts` would have recorded, and put the two side by side. You are looking for one thing: **behaviour in the fast trace that has no counterpart in the slow one.** Periodic structure faster than `2 x Ts` is the specific thing to look for, because that is what aliases.

Three practical notes.

Do it once per deployment site, not once per project. The 20-minute fan is a property of that greenhouse.

Do it again after anything changes near the sensor. New equipment, new ventilation, new neighbour.

Record the result in the credibility argument. "Sampled at 1 Hz for one hour on 2026-08-14; no periodic structure above 0.5 per minute observed" is a sentence Chapter 7's evidence section can use, and its absence is a gap a reviewer should find.

9.3.4 Sampled, event-driven, and the arrangement you usually want

Two ways to get data off a physical twin.

Sampled. Read every `Ts`, whether anything changed or not. Predictable volume, predictable load, trivially detectable gaps – a missing sample is visible because you know exactly when it should have arrived. Wasteful when nothing is happening, and blind between samples.

Event-driven. The device reports when something happens or when a value crosses a threshold. No waste, no blindness to the events it is configured for. But: **silence is ambiguous.** Nothing happened, or the link is down, or the device is dead. That ambiguity is fatal on its own, and the standard fix is a heartbeat – a periodic "I am here and nothing happened" – which is a sampled channel wearing a different hat.

The arrangement you usually want is both, and the demonstrator shows why without anybody having designed it that way:

- Moisture is *sampled*, because it changes continuously and slowly, and because Chapter 7's calibration needs regular observations at known times.
- Watering events are *events*. A dose is a discrete thing that happens at 09:16, and sampling for it would be absurd.

And the pairing is where the twin's value comes from. Chapter 1's whole fault detector compares an event (a dose was commanded) against a sampled consequence (the moisture step). Neither stream alone detects anything. This is a general shape worth carrying: **sample the state, capture the events, and align them on time** – which makes Sec. 9.5 the section everything depends on.

9.4 Protocols: what a twin needs from a transport

9.4.1 Two layers, and only one of them is your problem

The device layer. I2C, the Serial Peripheral Interface (SPI), MODBUS, 4-20 mA, and the Controller Area Network (CAN) bus. How a sensor talks to the thing next to it. In the demonstrator, the STEMMA sensors sit on I2C behind a multiplexer and the relay module speaks MODBUS. Characteristics: short range, no security, no notion of identity beyond an address, and usually somebody else's firmware between you and it.

The system layer. HTTP, the Message Queuing Telemetry Transport (MQTT), the Open Platform Communications Unified Architecture (written OPC UA), the Advanced Message Queuing Protocol (AMQP), the Data Distribution Service (DDS), Kafka. How the physical twin's software talks to yours. This is your layer.

The boundary between them is a component somebody owns, and it is worth finding out who. In the demonstrator it is the Raspberry Pi: it drives I2C and MODBUS, writes a local time-series database, and exposes HTTP on port 8099. **That Pi is a gateway**, and recognising it as one immediately raises the right questions – what happens when it reboots, whose clock is authoritative, how much does it buffer – none of which are visible if you think of it as "the sensor endpoint".

9.4.2 Poll versus subscribe

	Poll	Subscribe
Who initiates	You	The device or broker
Latency	Up to one interval	As fast as the path
Load	Predictable	Bursty
Connection state	Known every interval, for free	Needs a heartbeat or a broker feature
Missed data	Recoverable if the source keeps history	Gone, unless the broker persists
Complexity	Low	Moderate, and mostly in failure handling

The demonstrator polls, and the reason is not a preference. The physical twin offers HTTP GET and nothing else. Chapter 3 Sec. 3.7.1 made the point and it bears repeating as a rule: *you architect against the interface that exists.*

What makes polling work here is one feature of that API, and it is worth naming because you should look for its equivalent everywhere: `GET /sensing/{unit}/{parameter}` takes a `since_timestamp`. That single parameter turns polling from lossy into resumable. Store the last successfully ingested timestamp per binding, ask for everything after it, and a two-hour outage becomes a two-hour delay instead of a two-hour hole.

> **The question to ask of any source, before anything about protocol:** *can I ask you for what I missed?* A source that can only tell you its current value forces you to make every poll count, and a source with history lets you fail and recover. This distinction matters more than the choice of protocol, and it is the one most often left out of an interface specification.

Figure 9.3 is why that question outranks the protocol choice. The same two-hour outage produces two completely different outcomes.

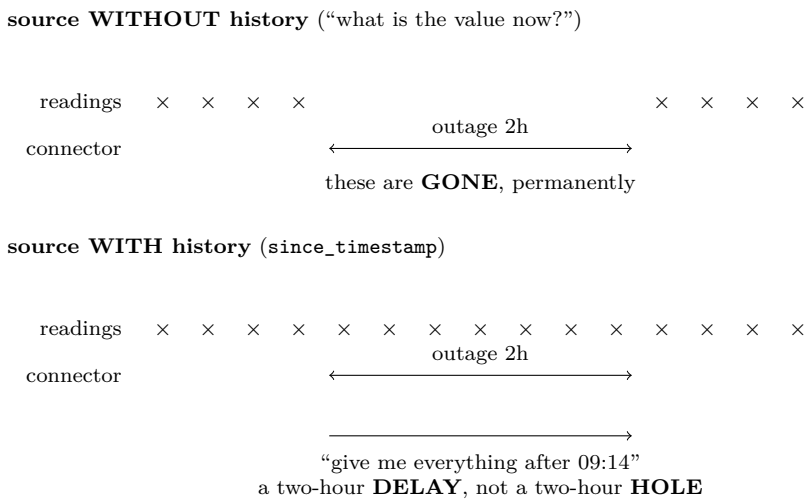

Figure 9.3 Resumability beats protocol choice. Poll or subscribe changes latency; history changes whether an outage costs you data at all.

The demonstrator's `since_timestamp` parameter is the whole difference between the two halves of that figure, which is why Sec. 9.7.1's treatment of gaps can be as short as it is.

9.4.3 Five properties a twin needs from a transport

Rather than compare protocols by feature, evaluate any transport against the five things a twin's downstream components actually depend on. Each row names which earlier chapter breaks if the property is absent.

Property	The question	Who breaks without it
Event time preserved	Does the reading carry when the physical thing was in that state, or only when the message arrived?	Chapter 5's replay, Chapter 6's estimator, Chapter 7's every comparison. This is the one to check first
Connection state observable	Can I tell "nothing happened" from "I am not receiving"?	Chapter 3's `quality`; Chapter 7's completeness; every alert that should have fired
Gap survivable	After an outage, can I get what I missed?	Chapter 7's calibration windows; Chapter 5's reproducible replay
Ordering knowable	If messages arrive out of order, can I tell?	Chapter 6's estimator, which is a sequential algorithm and does not tolerate a shuffled history
Delivery idempotent-friendly	Is there a stable key I can deduplicate on?	Chapter 7's counts, Chapter 8's training sets, and anything that averages

Read that table as a procurement checklist, because that is what it is for. Any transport can be made to satisfy all five with enough work at your end; the table tells you how much work, and where it lands. A source that publishes bare values with no event time and no sequence number is not cheap. It is expensive, and the expense is yours.

Note what is absent from the table: throughput, latency, security features, tooling, ecosystem. Those matter and they are real selection criteria – they are just not *twin* criteria, and Chapter 12 selects tools.

9.4.4 The names, for recognition only

You will meet these in a first meeting and should be able to place them. One line each, deliberately.

- **HTTP/REST** – request/response. Poll, and hope for a history parameter. Universal, and the demonstrator's.
- **MQTT** – lightweight publish/subscribe through a broker, built for constrained devices and unreliable links. The default in IoT-shaped systems.
- **OPC UA** – the industrial-automation standard, carrying an information model as well as data. The one that answers "what is this value?" as well as "what is it now?"
- **DDS** – publish/subscribe with no broker, aimed at low-latency distributed control.
- **AMQP** – brokered messaging with queueing semantics, from the enterprise side.

- **Kafka** – a durable, replayable log. Not a device protocol; frequently the thing behind the gateway.

Surveys of the publish/subscribe landscape exist and are candid that there is no common ground on which to compare these systems, because they differ in brokers, delivery policies, protocols and APIs [4]. Twin middleware studies find the same heterogeneity and treat it as the problem middleware exists to absorb [1]. Digital-twin platform guidance names DDS, MQTT and OPC UA together as the connectivity middleware set [2].

Chapter 12 selects. Chapter 13 covers the standards. This chapter's position is that the five properties of Sec. 9.4.3 decide more than the name does.

9.4.5 One handler per protocol, one normalised form upstream

The architectural conclusion is Chapter 3's and needs only a sentence of reinforcement: **put each protocol behind its own handler, and let the rest of the system see tuples.**

Two consequences you can act on immediately.

The handler is where the physical world's vocabulary dies. Register numbers, multiplexer channels, endianness, scaling factors, the fact that this device reports tenths of a degree as an integer – all of it stops here. A scaling factor that escapes the handler will eventually be applied twice.

The handler is the only place a unit is attached, and it must be. Chapter 3 put `unit` in the tuple. A connector that emits bare numbers has exported a convention, and conventions are not checkable.

9.5 Time: four timestamps and a clock

This is the section that Chapter 2 has been owed since Sec. 2.7, and the one where the most expensive quiet defects live.

9.5.1 There are four timestamps, and they are all different

Follow one moisture reading from soil to store:

#	Timestamp	Set by	Why it differs from the one before
1	**Sensor read time**	The gateway, when it reads the bus	The physical state was what it was; this is when somebody looked
2	**Gateway record time**	The Pi's local database	Decoding, bus contention, batching
3	**Transport time**	The broker or the HTTP response	Network, queueing, the poll interval itself

#	Timestamp	Set by	Why it differs from the one before
4	**Ingest time**	Your connector, on acceptance	Your processing, your retries, your backlog

Figure 9.1 puts them on one timeline, which is where the point becomes visible: the four are not alternatives, they are *stations on a path*, and the distance between the ends is a quantity you need.

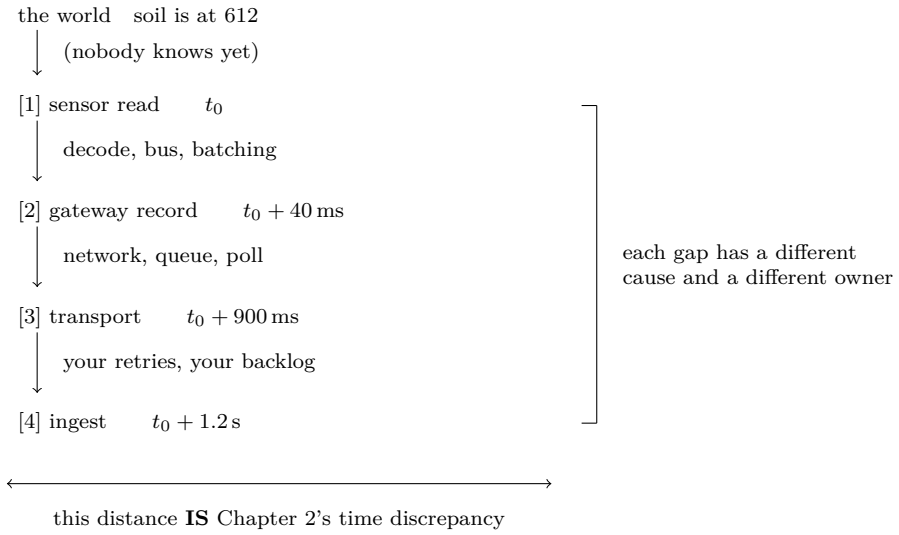

Figure 9.1 Four timestamps on one path. Picking one and discarding the rest throws away the only number that tells you how far behind the twin is running.

Chapter 3's tuple has **one** timestamp field. Which of the four goes in it?

> **Rule: the tuple carries event time – the earliest one that describes the world rather than the plumbing – and the connector separately records ingest time.**

Event time is what every model comparison in this book needs. Chapter 7 compares a prediction for 17:05 against a measurement at 17:05, and if the "17:05" is when your service happened to accept the message, the comparison is against a number from a different moment. Ingest time is what every operational question needs: how far behind are we, is the backlog growing, did this arrive late.

Carry both. They are not redundant, and their difference is a measurement – specifically, Chapter 2 Sec. 2.7.2's **time discrepancy**, which has been studied in its own right for twins of cyber-physical systems precisely because there are cases where the delay is the whole problem [5].

In the demonstrator, event time is the timestamp the Pi's API returns, which is timestamp 1 or 2 depending on how the Pi records it – and *finding out which* is a real

task with a real answer, not a detail. Ingest time is your connector's own clock when the poll succeeded.

9.5.2 The timestamp questions that catch people

Four, all cheap to answer at design time and expensive later.

Whose clock? The Pi's and yours are different clocks. If event time comes from the Pi and you compare it against your own "now", you are subtracting two clocks (Sec. 9.5.3).

What is the offset from UTC, and is it stored? Local time without an offset is not a timestamp. The failure is famous and it still happens, and in a twin it produces a one-hour residual bias twice a year that Chapter 7's machinery will faithfully attribute to a modelling assumption.

What resolution? A timestamp to the second is fine at `Ts` = 10 minutes and useless at 2 kHz. Match it to the sampling interval, not to a habit.

Is it the start or the end of the interval? For an instantaneous reading this is meaningless. For anything aggregated – a mean over a minute, a count over an hour – it decides whether your data is shifted by the window length. **Aggregates need both edges, or a stated convention that is actually stated.**

9.5.3 Clock error, sized against `Ts`

Two clocks drift apart. The question is never "is the clock accurate" but **"is the clock error small compared with `Ts`?"** Here is that comparison done twice, with different answers.

The arithmetic. An uncompensated crystal oscillator is commonly specified around 20 parts per million. That is 20 microseconds per second:

```
20 ppm x 86,400 seconds per day = 1.73 seconds per day
                    over a month = about 52 seconds
                     over a year = about 631 seconds, or 10.5 minutes
```

Case 1: the demonstrator. `Ts` = 10 minutes. A year of unsynchronised drift equals **one entire sampling interval** – so an unsynchronised gateway will, after a year, place readings a whole sample away from where they belong. Over a week, though, drift is about 12 seconds, which is 2 per cent of `Ts` and harmless. **Ordinary network time synchronisation, which typically holds tens of milliseconds, is four orders of magnitude better than needed here.** Turn it on, verify it is running, and stop thinking about it.

Case 2: the turbine. Vibration sampled at 2 kHz means a sample every 0.5 milliseconds. Now tens of milliseconds of clock error is *tens of samples*, and two accelerometers on different gateways cannot be compared at all. Ordinary network time is not enough; this is where precision time protocols and clocks disciplined by the Global Positioning System (GPS) earn their cost.

> **The rule, and it is the whole section:** express clock error as a fraction of `Ts`. Below a per cent or so, ignore it. Approaching `Ts`, your timestamps are fiction. The demonstrator and the turbine differ by six orders of magnitude in

`Ts` and therefore need completely different answers to a question that sounds identical.

One more, because it bites in exactly one place. A gateway that reboots without a battery-backed clock may come up at the epoch, or at its last known time, and then jump when it synchronises. Both produce readings with absurd or duplicated timestamps. **Your connector must reject an event time that is impossible** – before the physical twin was commissioned, or in the future – rather than storing it. Chapter 8 Sec. 8.6.2's input-coverage check is the same instinct applied one layer down.

9.5.4 Computing twinning rate, age of twin, and time discrepancy

Chapter 2 Sec. 2.7 introduced three metrics and said the taxonomy does not carry them. The connector produces all three, cheaply, and they should be on a dashboard from day one.

Metric	Computed as	For the demonstrator
Twinning rate	Tuples accepted per binding per unit time	Expect 6 per hour per moisture binding at `Ts` = 10 min
Age of twin	`now` minus the newest event time held, per binding	Should hover just above `Ts`; a rising value is the first symptom of almost everything
Time discrepancy	Ingest time minus event time, per tuple	Expect roughly half a poll interval on average; a growing distribution means a backlog or a slowing source

Three notes that make these useful rather than decorative.

Per binding, always. An aggregate age-of-twin across twelve pots hides one dead sensor completely. Chapter 3's binding is what makes the disaggregation possible, and this is the payoff.

Age of twin is the single best one-number health indicator a twin has. It responds to a dead sensor, a dead gateway, a dead network, a crashed connector and a full disk, and it needs no configuration beyond `Ts`. Twin-platform work treats the twinning rate and the age of twin as measurable properties for exactly this reason [6].

Alert on age of twin, not on the absence of data. "No data received" is not an event; nothing arrives to trigger it. Something must be watching the clock, and it must be outside the component that stopped working.

9.6 Quality: building the field Chapter 3 defended

Chapter 3 Sec. 3.2.1 put `quality` in the tuple and defended it in one sentence: a reading you are unsure of and a reading you did not get are different, and a connector mapping both to `null` has destroyed information the state estimator needed. Here it is built.

9.6.1 The states, enumerated

Six is enough for most twins, and fewer than four is always too few.

State	Meaning	Value present?
`good`	Read successfully, inside range, source healthy	Yes
`uncertain`	Read, but something is off – near a range limit, sensor overdue for calibration, unusually noisy neighbourhood	Yes, and it must not be silently used as `good`
`substituted`	Not measured; produced by the twin – interpolated, forward-filled, or estimated	Yes, **and this is the state that must never be lost**
`out_of_range`	Read, but outside the sensor's specified range	Yes, but as evidence about the sensor, not about the world
`no_reading`	The attempt was made and failed	**No**
`not_attempted`	No attempt was made – connector down, scheduled outage	**No**

Why `substituted` is the important one. Every twin eventually fills a gap, and filling gaps is legitimate. What is not legitimate is a filled gap that looks like a measurement. Chapter 7's hold-out spread of 2.0 would be meaningless if a third of the readings were forward-filled, and Chapter 8's learned correction would be trained partly on its own outputs. **A substituted value must be as straightforward to exclude as it was to create**, and that is a schema decision made here, in the connector, before anybody has a reason to want it.

Why `no_reading` and `not_attempted` are separate. One says the physical twin failed to answer; the other says you did not ask. They point at different teams and different fixes, and a completeness calculation (Sec. 9.7.1) that cannot tell them apart will blame the wrong one.

9.6.2 What each state does downstream

A quality state that nothing consumes is decoration. Each of these has a defined consumer, and specifying that is part of building the connector.

State	Estimator (Ch6)	Calibration / hold-out (Ch7)	Alerting (Ch11)	Learned model (Ch8)
`good`	Use normally	Include	Normal	Include
`uncertain`	Use with reduced gain	**Exclude**	Normal, with a flag	Exclude
`substituted`	Do not correct toward it	**Exclude**	Suppress	**Exclude, always**
`out_of_range`	Ignore the value; count the event	Exclude	Alert on the sensor	Exclude
`no_reading`	Predict without correcting	Mark the window incomplete	Feeds the gap detector	Gap
`not_attempted`	Predict without correcting	Mark the window incomplete	Suppress	Gap

Two rows are worth pausing on.

`uncertain` *reduces the estimator's gain.* Chapter 6 Sec. 6.6.2 defined the gain as how far the correction pulls toward the measurement, and Chapter 7 Sec. 7.7.4 made a drifting gain an expiry trigger. This is the mechanism by which a doubtful reading has a proportionate effect instead of a binary one, and it is available only because the connector said "unsure" instead of "here is a number".

Three states exclude from calibration. Chapter 7's campaigns assumed complete, trustworthy windows and never said so. This table is where that assumption becomes enforceable.

9.6.3 Where quality is decided, and where it cannot be

Most of it is decided in the connector, because most of it is knowable there: the read failed, the value is outside the datasheet range, the sensor is overdue for calibration, the value was substituted.

One thing is not, and it is worth being clear about the boundary. Whether a reading is *implausible given what the twin knows* cannot be decided by the connector, because the connector has no model. A moisture reading of 640 is inside range and perfectly good; a moisture reading of 640 ten minutes after a reading of 380 is nearly impossible, and only something holding Chapter 4's model can say so. Sec. 9.7.4 finds the same boundary from the other direction.

So the architecture is: **the connector decides quality it can determine from the sensor and the transport; a downstream check decides quality that requires the model.** Keep them separate, and do not let the connector grow a model – that is a different component with a different lifecycle, and Chapter 3 drew it as one.

Broader treatments of data quality for IoT-shaped systems make the same distinction between quality of data and quality of information, and treat the evaluation of autonomic applications as a problem in its own right [7].

9.7 The four ways a stream goes wrong

Every measurement stream fails in four ways. Three are handled by mechanisms you already have; the fourth defeats all of them.

9.7.1 Gaps

What it is. Expected readings did not arrive.

Detection, and it is not "no data received". Nothing arrives to trigger an event, so a gap must be detected by something watching the clock. Two checks, and both are needed:

Live check – age of twin. Alert when `now` minus the newest event time exceeds `k x Ts`. For the demonstrator, `k` = 3 gives 30 minutes. Why not `k` = 1: a single missed poll is ordinary and alerting on it trains people to ignore alerts. Why not `k` = 10: 100 minutes is longer than a moisture step takes to resolve, so a fault could be missed entirely within one alerting window.

Retrospective check – completeness. Count what arrived against what was expected:

```
expected per day at Ts = 10 min:  24 x 6 = 144 samples per binding
completeness = received / expected
```

A day with 130 of 144 is 90 per cent complete. Pick a threshold and act on it:

> **Mark a window degraded when completeness falls below 90 per cent, and exclude degraded windows from calibration and validation.**

That single rule closes a hole in Chapter 7. Its calibration campaign used six nights and its hold-out used week 3, and neither said anything about whether those windows were complete. A night with a two-hour hole in it produces a drying rate computed across a gap, and the arithmetic will not notice.

Handling. Backfill where the source supports it – `since_timestamp` in the demonstrator's case. Where it does not, emit `no_reading` and leave the hole. **Do not forward-fill into the measurement stream.** If something downstream needs a value at every tick, it may substitute one, and it marks it `substituted`.

9.7.2 Duplicates

What it is. The same reading ingested more than once. Caused by retries, at-least-once delivery, an overlapping backfill, or two connector instances.

Why it matters more here than in most systems. Duplicates do not look like errors. They look like extra data, and extra data quietly reweights every average, every count and every fit. Chapter 7's six-night average and Chapter 8's training set are both silently wrong if one night was ingested twice.

Handling, and this is the tractable one. Make the write idempotent on a natural key – `(binding, quantity, event_time)`. An upsert on that key absorbs retries, restarts and overlapping backfills without any coordination. Chapter 3 Sec. 3.7.1 already specified

this for the demonstrator; the point here is that **it is a property of the key, not of the protocol**, and it is therefore available to you no matter what the source offers.

The complication worth knowing. If two genuinely different readings share a key – two sensors on the same binding, or an event-time resolution coarser than the sample interval – the upsert silently discards one. **Verify that your key is actually unique in your data before relying on it**, which is a five-minute query and a real source of quiet loss.

9.7.3 Out-of-order arrival

What it is. A reading for 10:00 arriving after one for 10:10. Normal with retries, backfills, multiple gateways, or any queue with parallel consumers.

Who cares, and it is not everyone. A store keyed on event time does not care at all – rows land where they belong. The consumers that care are **sequential** ones: Chapter 6's estimator runs a predict-correct cycle in order and cannot accept a correction from the past after it has already moved on.

Handling: pick one, and write down which.

- *A watermark.* Process up to `now - w` and treat anything older as late. Simple, and costs `w` of latency.
- *Recompute.* Accept late data, and re-run the affected estimator steps. Correct, and requires the runner to be re-runnable – which Chapter 5 Sec. 5.6's reproducibility requirement already gives you, if you built it.
- *Drop and count.* Legitimate when late data cannot change a decision. **Only legitimate if the count is monitored**, because a rising drop rate is a real problem wearing a working system's clothes.

For the demonstrator, a watermark of one poll interval is right, and the reason is that nothing acts within 10 minutes anyway.

9.7.4 The stuck sensor, which defeats everything above

What it is. The sensor reports an unchanging, entirely plausible value. No gap. No duplicate. No ordering problem. Perfect completeness. Every mechanism in this chapter reports a healthy stream.

Figure 9.2 is why: the three failures earlier in this section all leave a mark on the stream, and this one leaves none.

```
gap            x x x . . . x x x      a hole you can see
duplicate      x x x x x x x x        two identical arrivals
                   ^^                  with the same event time
out-of-order   x x   x x x            arrival order != event
                   ^                   order

ALL THREE are visible in the stream's SHAPE.


stuck          x x x x x x x x x x    perfectly regular
               6 6 6 6 6 6 6 6 6 6    perfectly plausible
```

```
                1 1 1 1 1 1 1 1 1 1    complete
                2 2 2 2 2 2 2 2 2 2

  the stream is healthy by every check in Sec. 9.7.1-9.7.3.
  The only thing that knows is the MODEL:

      expected change per sample = 0.71 units
      sensor resolution          = 1 unit
      -> two identical readings prove nothing
      -> but over an hour, the model expects 4.28
```

Figure 9.2 Why the stuck sensor is the one to fear. The other three failures are detectable from the stream; this one is detectable only against a model of what the value should have done.

This is the failure mode to be afraid of, and it is common: a frozen firmware register, a cached value in a driver, a sensor whose element has detached, an analogue input that has settled.

Detection starts as arithmetic. A stuck sensor is one whose value does not change when the model says it should. Chapter 7 fitted `k_day` = 4.28 reading units per hour, so over one 10-minute sample:

```
expected change per sample = 4.28 / 6 = 0.71 reading units
```

The sensor reports whole numbers, so its resolution is 1 unit. **The expected change per sample is smaller than the resolution** – which means two identical consecutive readings are entirely ordinary and cannot be evidence of anything. Over an hour, though:

```
expected change per hour = 4.28 reading units
```

which is four resolution steps and comfortably visible. So:

> **Flag a moisture sensor as suspect when its value is unchanged across six consecutive samples (one hour) during a period in which the model expects a change of at least a few resolution steps.**

And now the trap, which is the reason this subsection exists. Chapter 4's assumption A2 says the sensor flattens near saturation: a pot watered to the top of the range shows no change *because the physics is real and the sensor is at its limit.* A naive unchanging-value detector fires precisely then – on a healthy sensor, in a condition Chapter 4 predicted in writing, four chapters before anybody wrote this code.

So the rule has that clause in it: *during a period in which the model expects a change.* Which means:

> **The stuck-sensor detector needs the model, so the connector cannot implement it alone.**

That is a genuine qualification to Chapter 3's clean layering, and it is worth stating rather than smoothing over. The connector can flag the raw symptom – "unchanged for N samples" – as `uncertain`. Deciding whether that symptom means a broken sensor

requires the model, so it belongs to a downstream check that has one. Chapter 8 Sec. 8.3.1 already described the right home for it: an anomaly detector running on model residuals rather than on raw signals.

9.7.5 The four, as a table you can build against

Failure	Symptom	Detected by	Handled by	Cost of missing it
Gap	Nothing arrives	Age of twin; completeness count	Backfill, or `no_reading`	Silent holes in calibration windows
Duplicate	Extra data	Uniqueness of the natural key	Idempotent upsert	Quietly reweighted averages and fits
Out of order	Event time before the last processed	Watermark comparison	Watermark, recompute, or counted drop	A corrupted sequential estimator
Stuck	Nothing wrong at all	**Model residual**, not the connector	`uncertain`, then a downstream check	Confident wrong decisions, indefinitely

Read the last column. The three a connector can detect have costs measured in bad data. The one it cannot has a cost measured in bad decisions, and that ordering is not a coincidence – the failures a system can see are the ones it was built to see.

9.8 Volume, buffering, and the link you do not own

Three decisions that follow from the ones above, and one warning.

9.8.1 Data volume, computed before it is a problem

Do this arithmetic at design time. It takes five minutes and it changes architectures.

The demonstrator. Say twelve plants – the rig's plant count is a deployment choice, and the conclusion below survives any of them – with one moisture quantity each, plus temperature and humidity from the SHT45 and eight spectral channels from the AS7341:

```
quantities        = 12 + 2 + 8 = 22
samples per day   = 22 x 144 = 3,168 tuples
at ~120 bytes each = about 380 KB per day
                   = about 139 MB per year
```

That number is the point. A hundred and thirty-nine megabytes a year means no architecture decision is forced by volume. Do not build a streaming pipeline for this. Chapter 12 will offer you platforms sized for a thousand times more, and the arithmetic above is your defence.

Now change one thing. Raise the rate to 1 Hz because somebody said "real time":

```
22 x 86,400 = 1.9 million tuples per day
            = about 228 MB per day
            = about 83 GB per year
```

Six hundred times the data, for a decision that happens twice a day. Chapter 2 called "real time" an abdication; this is the invoice.

And the other extreme – the turbine. Three triaxial accelerometers at 2 kHz, four bytes per sample:

```
9 channels x 2,000 samples/s x 4 bytes = 72,000 bytes per second
                                       = about 6.2 GB per day per turbine
                                       = about 2.3 TB per year per turbine
an 80-turbine farm                     = about 180 TB per year
```

Now the architecture is forced, and this is where edge preprocessing stops being an optimisation and becomes the design: extract features at the turbine, ship the features, keep raw windows only around events. Structural monitoring architectures are built this way for exactly this reason [8], [9], and the twin sensing literature treats compression as a first-class concern for the same constraint [3].

> **The general lesson.** The sampling interval is a capital expenditure decision. Sec. 9.3.1 derived `Ts` from the decision the twin serves; this subsection prices the answer. Do both before anybody buys anything.

9.8.2 Buffering and store-and-forward

The question. When the link between the physical twin and your connector breaks, does the physical twin hold on to its data?

If yes, an outage is a delay. When the link returns, backfill. Everything in Sec. 9.7.1 works, and the demonstrator is in this happy state because the Pi keeps a local time-series database and the API takes `since_timestamp`.

If no, an outage is a hole, permanently. And then the honest engineering response is either to add buffering at the physical-twin end, or to accept the holes and put them in the credibility argument – because Chapter 7's limitations section is where an un-fixable gap belongs.

Three practical notes.

How much buffer? Enough for your worst realistic outage. If the site loses connectivity for a day twice a year, a two-day buffer converts an annual data-loss incident into an annual non-event.

Buffering is not free at the far end either. A device that has buffered six hours will deliver six hours at once, and your ingest must survive the burst – which is Sec. 9.8.3.

Where the buffer sits is a topology decision, and Chapter 12 owns it. Work on constrained devices paired with twins that offload computation and validate inferences is one shape this takes [10]; a gateway with a disk is another and is usually enough.

9.8.3 Backpressure, and the burst you did not test for

A connector that keeps up in the steady state is not a connector that keeps up. The load that breaks it is the backfill: six hours of buffered data arriving in one response, or twelve bindings all recovering at once after a network event.

Two rules that cost nothing at design time.

Bound every queue, and decide what happens when it is full. An unbounded queue does not prevent loss; it converts fast, visible loss into slow, invisible loss ending in an out-of-memory kill. A bounded queue with a counted drop is worse-looking and better.

Make backpressure reach the poller. A polling connector controls its own input rate, which is a real advantage over a subscription, and it is thrown away by accident more often than not: if the poll loop runs on a timer regardless of whether the previous batch has been written, a slow store turns into an unbounded backlog. Poll, write, then schedule the next poll.

Test it deliberately. Stop the connector for two hours, restart it, and watch. This is a fifteen-minute test that finds a class of defect nothing else finds, and it is the same test that verifies Sec. 9.7.1's backfill and Sec. 9.7.2's idempotency at the same time.

9.8.4 The link is an attack surface, and it is not this chapter's

Two things belong here and the rest is Chapter 13's.

What is different about this layer. The acquisition and dissemination path is where the twin's contact with the physical world is thinnest and most exposed. A comprehensive threat survey classifies twin threats by functional layer and notes that a loss of integrity or availability in the data-dissemination layer propagates – affecting synchronisation services and the quality of the twin's simulations, not merely the data [11]. That propagation is the reason a connector-level compromise is not a connector-level problem.

What you can do here, cheaply. Everything in this chapter that rejects implausible input does double duty: rejecting impossible event times (Sec. 9.5.3), rejecting out-of-range values (Sec. 9.6.1), and Chapter 8 Sec. 8.6.2's input-coverage check all constrain what an unexpected input can do. None of that is a security control, and none of it substitutes for Chapter 3 Sec. 3.3.3's. It is the part you get for free by building the connector well.

9.9 Worked example: the demonstrator's connector, built

Chapter 3 Sec. 3.7.1 gave this component eight lines and a decision. Here it is in full, with every choice traced to a section above.

9.9.1 What the source offers, read as a specification

From Chapter 1 Sec. 1.8.1:

- `GET /sensing/{unit}/{parameter}`, with `limit` and `since_timestamp`
- `GET /actuation/{unit}/watering_events`

- HTTP on port 8099; no subscription, no push, no notification

Against Sec. 9.4.3's five properties:

Property	Available?	What that costs us
Event time preserved	Yes, the API returns timestamps	Nothing – but we must confirm whether it is sensor-read or database-write time
Connection state observable	Yes, for free – a poll either succeeds or does not	Nothing
Gap survivable	**Yes**, via `since_timestamp`	Nothing. This is the single most valuable feature of this API
Ordering knowable	Yes, event time is present and the source is one gateway	Nothing
Idempotent-friendly	Yes – `(binding, quantity, event_time)` is a stable key	Nothing

Five out of five, from a plain REST API with a history parameter. Worth noticing before reaching for anything more sophisticated: the properties that matter are not the properties that sound advanced.

9.9.2 Bindings and the tuple mapping

One binding per measured thing, not per device:

Binding	Quantity	Unit	Source
`pot-3/moisture`	soil moisture	reading units	`/sensing/pot-3/moisture`
`greenhouse/air_temperature`	air temperature	degrees C	`/sensing/greenhouse/air_tempe`
`greenhouse/humidity`	relative humidity	per cent	`/sensing/greenhouse/humidity`
`greenhouse/light_{n}`	irradiance, channel n	sensor units	`/sensing/greenhouse/light` (one poll, many tuples)
`pot-3/watering`	dose delivered	millilitres	`/actuation/pot-3/watering_eve`

Three decisions embedded there. The light sensor's one response becomes several tuples, per Sec. 9.2.3. The greenhouse bindings are shared across pots, so the store must join them by time rather than by pot – Chapter 10's problem, created here. And the watering stream is ingest, not actuation: this connector never commands a pump.

9.9.3 The poll loop

```
for each binding:
    last = watermark[binding]              # last event time successfully written
```

```
try:
    rows = GET /sensing/{unit}/{param}?since_timestamp=last
except any transport failure:
    emit (binding, quantity, -, -, now, no_reading)
    mark connector disconnected for this binding
    continue                              # do NOT advance the watermark

for row in rows:
    t = parse_event_time(row)             # with offset; reject if impossible
    v = decode(row)                       # units applied here and nowhere else
    q = classify_quality(v, t, binding) # Sec. 9.6.1
    upsert (binding, quantity, v, unit, t, q)   # key: binding+quantity+t
    record ingest_time = now

watermark[binding] = max event time written
schedule next poll                        # AFTER the write, per Sec. 9.8.3
```

Eleven executable lines, and every one of them is a section of this chapter:

- The watermark is Sec. 9.4.2's resumability, and it advances **only on successful write**, which is what makes a crash mid-batch safe.
- The failure branch emits `no_reading` rather than a zero – Chapter 3 Sec. 3.7.1's rule, and Sec. 9.6.1's state.
- `parse_event_time` rejects the impossible, per Sec. 9.5.3's rebooted-clock case.
- Units are applied inside the handler and nowhere else, per Sec. 9.4.5.
- The upsert key makes retries and overlapping backfills harmless, per Sec. 9.7.2.
- Scheduling after the write, not on a timer, is Sec. 9.8.3's backpressure rule and the difference between a delay and an unbounded backlog.

9.9.4 The numbers, all of them

Quantity	Value	Where it came from
`Ts`	10 minutes	Chapter 3 Sec. 3.7.1, re-derived by Sec. 9.3.1's five steps
Expected samples per binding per day	144	`24 x 6`
Gap alert	age of twin above 30 minutes	Sec. 9.7.1, `k` = 3
Completeness threshold	90 per cent, or 130 of 144	Sec. 9.7.1
Watermark lateness allowance	one poll interval	Sec. 9.7.3
Stuck-sensor symptom	unchanged across 6 samples	Sec. 9.7.4, from `k_day` = 4.28 and 1-unit resolution
Clock requirement	network time sync, verified running	Sec. 9.5.3, error under 1 per cent of `Ts`
Data volume	about 139 MB per year	Sec. 9.8.1

9.9.5 What this connector still cannot do, stated for Chapter 7

The credibility argument gets a paragraph out of this section, and it is an honest one:

> Measurements are polled every 10 minutes from the physical twin's HTTP API, with resumable backfill. Event time originates on the gateway; gateway and twin clocks are network-synchronised, with expected error below 1 per cent of the sampling interval. Windows below 90 per cent completeness are excluded from calibration and validation. Substituted values are marked and never used for fitting. **The connector cannot detect a sensor reporting a plausible unchanging value; that check requires the model and runs downstream. No check for aliasing has been performed at this site.** Watering events are ingested from the actuation history; the twin issues no commands.

The two bolded sentences are the ones a reviewer should find, and they are there because Sec. 9.7.4 and Sec. 9.3.3 said so. **A connector chapter that produced no new limitations for Chapter 7 would not have been honest about what a connector is.**

9.10 Faded example: the offshore turbine's ingest

Chapters 4 through 8 each took the turbine one step further. Now connect it. Two parts are worked; four are yours.

The system, recapped. A floating offshore wind turbine with a reduced-order structural model, a Kalman filter estimating loads from a handful of sensors, a fatigue service, and Chapter 8's anomaly detector on the estimator's residuals. Real twins of this shape are validated against full-scale prototype measurements [12], and diagnostic implementations on operational floating turbines exist [13].

(a) `Ts` is not one number here – worked, because it is the finding. The demonstrator has one sampling interval. This asset has at least three, and trying to pick one is the mistake:

Signal	Rate	Derived from
Vibration and strain	2 kHz	The structural dynamics being observed – the fastest thing that must be seen
Operational state: power, pitch, yaw, wind	1 Hz	Control-system update rate; enough for the estimator
Environment: sea state, met data	10 minutes	Industry convention and the rate at which it changes

And a fourth number that is not a sampling interval at all: the fatigue service integrates over years and consumes daily aggregates. Chapter 5's distinction between `h` and `Ts` was one collision of timescales; this is four, and the connector is where they meet. The design consequence is that **the high-rate streams almost certainly do**

not leave the turbine intact – Sec. 9.8.1's arithmetic gives 2.3 TB per turbine per year for vibration alone – so the connector's boundary sits at the edge, and what crosses it is features plus event-triggered raw windows.

(b) Clocks – worked, because the demonstrator's answer is wrong here. Sec. 9.5.3 concluded that ordinary network time sync is four orders of magnitude better than the pot needs. At 2 kHz the sample period is 0.5 milliseconds, and tens of milliseconds of network-time error is *tens of samples*. Two accelerometers on different acquisition units cannot then be compared, which destroys any analysis depending on the phase relationship between them – which is most structural analysis. This asset needs a precision time protocol or GPS-disciplined clocks, and that cost belongs in Chapter 1's estimate rather than in a surprise.

Now yours.

(c) Build the quality-state table (Sec. 9.6.1) for the vibration stream. Which of the six states apply, which do not, and what does `substituted` mean for a 2 kHz signal? *Hint: consider what a downstream feature calculation does with a gap, and whether interpolating 200 samples is ever the right answer.*

(d) Sec. 9.7.4's stuck-sensor detector used the model's expected change. Design the equivalent for an accelerometer. *Hint: a working accelerometer on a moving structure is never still, so the symptom is not "unchanging" – what is it, and what does it look like when the structure genuinely is becalmed?*

(e) Apply Sec. 9.4.3's five properties to a satellite or cellular link with intermittent connectivity and a hard bandwidth cap. Which property is hardest to satisfy, and what does buffering at the turbine (Sec. 9.8.2) cost in disk and in latency for the maintenance decision?

(f) The anomaly detector of Chapter 8 runs on residuals, which requires measurement and model output to be aligned in time. Given (a)'s three rates and (b)'s clock requirement, specify what the connector must guarantee for that alignment to be sound, and one thing that could go wrong in a way nothing downstream would notice.

9.11 Posed problem: ingest for a source you do not control

No solution is given.

The situation. You are building a twin of a district heating network: about 400 substations, each with a flow meter, supply and return temperature sensors, and a valve position. The meters are from three vendors installed over fifteen years.

- Vendor A's units (about 250) push readings over MQTT to a broker the utility runs, every 15 minutes, with a device-local timestamp. Roughly 2 per cent of devices are known to have wrong clocks.
- Vendor B's units (about 120) are polled over MODBUS Transmission Control Protocol (TCP) by a Supervisory Control and Data Acquisition (SCADA) system, which exposes an hourly CSV export. No history endpoint; the CSV is the interface.
- Vendor C's units (about 30, the oldest) are read manually by a technician every three months.

The twin must detect a failed heat exchanger, which shows as a persistent change in the difference between supply and return temperature.

Produce an ingest design of no more than four pages containing:

1. **`Ts` per source**, derived by Sec. 9.3.1's five steps from the detection decision – and a statement of whether Vendor C's substations can be twinned at all. *If your answer is that they cannot, say so; Chapter 1 Sec. 1.7 permits that answer and this problem is partly a test of whether you will give it.*
2. **The five properties of Sec. 9.4.3 assessed per vendor**, with the cost of each missing property assigned to a component you will have to build.
3. **A timestamp policy**, per Sec. 9.5, covering the 2 per cent of Vendor A devices with wrong clocks. State how you detect them and what you do with their data – and note that "discard" and "trust" are both defensible and have different consequences for the fault detector.
4. **The quality states** you will emit, per Sec. 9.6, including what Vendor B's hourly CSV means for `not_attempted` versus `no_reading`.
5. **The four failure modes** of Sec. 9.7, with detection for each per vendor. Pay particular attention to the stuck detector: what does the model expect the supply-return difference to do, and when does it legitimately not change?
6. **Volume arithmetic** per Sec. 9.8.1, and a statement of whether it forces any architectural decision.
7. **A completeness policy** and its consequence for Chapter 7: how much data must a substation have before you will calibrate a model for it?
8. **One paragraph for the credibility argument** naming what this ingest design cannot deliver.

What a good answer looks like. It treats the three vendors as three connectors behind one normalised form, not as one connector with branches. It notices that a fault detector on a *difference* between two sensors has a failure mode neither sensor has alone – both drifting together hides it, and one drifting alone fakes it. It says plainly that Vendor C's three-month cadence cannot support the decision, and either proposes instrumenting those 30 substations with the cost stated, per Sec. 9.2.4, or scopes them out. And it does the volume arithmetic and concludes it is small, rather than assuming 400 substations is a big-data problem.

9.12 Summary

Seven things, tied to the five objectives.

1. **The connector is a small contract and a large amount of work** (Sec. 9.1). One tuple type, one honest connection state – and every failure in the physical world arrives through it, which is why its cost is underestimated and why the twin's credibility begins here.
2. **A sensor is a model with a datasheet** (Sec. 9.2). Six numbers change your design, resolution and accuracy are not the same number, and a model's residual spread cannot beat its instrument's accuracy. A sensor is worth adding when it changes a decision, not when it adds a quantity – and its cost is the decoder, the

binding, the calibration, the failure modes and the disagreement policy, not the sensor. *(Objective 1.)*

3. **`Ts` is derived from the decision in five steps, and the derivation can still alias** (Sec. 9.3). A ventilation fan on a 20-minute cycle sampled every 10 minutes produces a clean trend that is not happening. The check is empirical, costs an afternoon, and belongs in the credibility argument. *(Objective 2.)*
4. **Evaluate a transport on five properties, not on features** (Sec. 9.4): event time preserved, connection state observable, gap survivable, ordering knowable, deduplication possible. The demonstrator's plain REST API scores five out of five because it has a history parameter, which is worth more than any protocol feature. *(Objective 3.)*
5. **There are four timestamps and two clocks** (Sec. 9.5). The tuple carries event time, the connector records ingest time, and their difference is Chapter 2's time discrepancy. Size clock error as a fraction of `Ts`: an unsynchronised 20 ppm clock drifts 10.5 minutes in a year, which is one whole sampling interval for the pot – while for the turbine, sampling at 2 kHz, even well-behaved network time synchronisation leaves tens of samples of error, which is why that asset needs a different answer to the same question. Age of twin, per binding, is the best single health metric a twin has. *(Objective 4.)*
6. **Quality is six states with defined downstream consumers** (Sec. 9.6). `substituted` is the one that must never be lost, `uncertain` is the one that reduces the estimator's gain, and three of the six exclude a window from calibration – which is where Chapter 7's unstated assumption of complete data becomes enforceable. *(Objective 5.)*
7. **Four failures, and the fourth is the dangerous one** (Sec. 9.7). Gaps, duplicates and out-of-order arrival are detectable in the connector and cost you bad data. A stuck sensor is invisible to every one of those mechanisms and costs you bad decisions – and detecting it requires the model, so it cannot live in the connector. That is a real qualification to Chapter 3's layering and this chapter states it rather than smoothing it. *(Objective 5.)*

And the note this chapter adds to the rest of Part III. Every number Part II reasoned about – the residual, the spread, the hold-out, the gain, the training set – is a number this component produced. Part II asked whether the twin deserves to be believed. Chapter 9's answer is that a large part of that question is settled before any model runs, by a piece of software nobody puts on a slide.

9.13 Exercises

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

9.13.1 *(Objective 1.)* A datasheet gives: range 0-100 per cent, resolution 0.1 per cent, accuracy plus or minus 3 per cent, drift 1 per cent per year, response time 30 seconds, maximum sample rate 1 Hz. Your model predicts changes of about 0.5 per cent per hour and your decision needs to detect a change of 2 per cent. State (a) the shortest interval over which a real change is distinguishable from instrument accuracy, (b) whether the resolution figure is useful, and (c) what the drift figure implies for Chapter 7's expiry conditions.

9.13.2 *(Objective 1.)* Your colleague proposes adding a second temperature sensor to a tank "for redundancy". Using Sec. 9.2.4's four steps, list the five costs beyond the sensor, and name the one question that must be answered before the second sensor is worth anything at all.

9.13.3 *(Objective 2.)* A pump runs on a duty cycle of 4 minutes on, 4 minutes off. You sample its outlet pressure every 8 minutes. Describe what your data will show, why it is worse than showing nothing, and give two sampling intervals that would not have this problem – one obvious and one that works for a different reason.

9.13.4 *(Objective 2.)* Derive `Ts` by Sec. 9.3.1's five steps for a twin that must warn a technician within 15 minutes that a cold-store door has been left open, given that the internal temperature rises about 2 degrees per hour with the door open and the sensor is accurate to plus or minus 0.5 degrees.

9.13.5 *(Objective 3.)* Assess these three sources against Sec. 9.4.3's five properties: (a) an MQTT topic publishing a bare number with no timestamp; (b) a REST endpoint returning the current value only, with no history; (c) a Kafka topic with event-time keys and seven days of retention. For each missing property, name the component you must build.

9.13.6 *(Objective 4.)* A gateway's clock is specified at 50 ppm and is never synchronised. Compute the drift after one week, one month and one year. Then state whether it is acceptable for (a) $\texttt{Ts} = 1$ hour, (b) $\texttt{Ts} = 1$ second, (c) a system correlating two such gateways at $\texttt{Ts} = 1$ second.

9.13.7 *(Objective 4.)* Your twin's age of twin for one binding is normally 11 minutes and has risen to 46 minutes over two days, with completeness still at 100 per cent. Give three possible causes consistent with both facts, and the one measurement that distinguishes them.

9.13.8 *(Objective 5.)* For each of the six quality states in Sec. 9.6.1, give one concrete cause in the demonstrator, and say what the Kalman filter of Chapter 6 should do with it.

9.13.9 *(Objective 5.)* A colleague says: "we forward-fill gaps at ingest so downstream code never has to handle missing values – it's simpler." Write the two-sentence objection, and name the specific number in Chapter 7 that this practice would invalidate.

9.13.10 *(Objectives 1-5, and the one that leaves the book.)* Point a poller at the real plant-controller API. Run it for a week. Then report: observed completeness per binding, the distribution of ingest time minus event time, whether the API's timestamp is sensor-read or database-write time and how you found out, and one thing about the real stream that this chapter did not prepare you for.

Solutions and hints

9.13.1. (a) A real change of 2 per cent at 0.5 per cent per hour takes **4 hours**, and instrument accuracy is plus or minus 3 per cent – so a *single pair* of readings can never establish it, because the change you care about is smaller than the accuracy. You must either average many readings within each window (accuracy that is random averages down; bias does not) or accept a longer detection time. **This is the most important row in the exercise** and it is the reason Sec. 9.2.2 says accuracy puts a floor under Chapter 7's

spread. (b) The resolution of 0.1 per cent is useful *for detecting change* – small changes are visible – but the two extra digits it offers relative to accuracy are decoration in any absolute statement. (c) 1 per cent per year against a 2 per cent decision threshold means the sensor consumes half your detection margin in a year. That is a recalibration schedule, and it is an expiry condition for Chapter 7's argument.

9.13.2. The five costs: a second decoder or channel; a second binding and its identity management; a second calibration and its ongoing recalibration; a second set of failure modes to detect and handle; and a **disagreement policy**. The question that must be answered first: *what will you do when they disagree?* If the answer is "look at both and decide", there is no redundancy – there is a second number and a human. Redundancy requires a rule decided in advance, and with two sensors you cannot vote, which is why redundancy schemes usually come in threes.

9.13.3. Sampling every 8 minutes against a 4-on/4-off cycle means every sample lands at the same phase: you will record a **constant pressure**, and which constant depends on when you started. It is worse than showing nothing because it looks like a healthy steady signal, and both a human and a model will believe it. Two intervals that avoid it: **much faster** – 30 seconds gives about 8 samples per half-cycle, the Sec. 9.3.1 convention – and, for a different reason, **deliberately non-uniform or prime-ish sampling**, for example every 70 seconds, which walks through the phase instead of locking to it. The second answer is worth knowing because it is sometimes the only one available when the fast rate is unaffordable; note that it complicates every gap and completeness check in Sec. 9.7.

9.13.4. (1) Decision and latency: warn within 15 minutes. (2) Physical event: temperature rising at 2 degrees per hour, i.e. 0.5 degrees per 15 minutes. (3) See it several times: a detectable change must clear the accuracy of plus or minus 0.5 degrees, so a single sample pair 15 minutes apart is marginal – 0.5 degrees of signal against 0.5 degrees of accuracy. Sample every **2 to 3 minutes** so that a rising trend across 5 to 7 samples is visible well inside the budget; a trend across several samples is detectable when a single difference is not. (4) Ceiling: trivial for any temperature sensor. (5) Floor: satisfied with margin. **The lesson is step 3**: the naive answer, "sample every 15 minutes because the requirement is 15 minutes", produces a detection that arrives at the deadline with no evidence.

9.13.5. (a) Fails *event time preserved* and *idempotent-friendly*; ordering only as good as the broker. You must build: a timestamping layer at the subscriber (which records ingest time and calls it event time – state that as a limitation, do not hide it) and a dedupe scheme with no natural key, which usually means content hashing plus a window. (b) Fails *gap survivable* entirely and, in practice, *event time preserved*. You must build a poller whose every miss is permanent, and put the resulting holes in the credibility argument. This is the demonstrator's API *without* `since_timestamp`, and comparing the two is the point of the exercise. (c) All five, with a seven-day bound on gap survival – so you must build nothing, but you must state the recovery window, because an eight-day outage is a permanent hole and a seven-day one is not.

9.13.6. 50 ppm = 50 microseconds per second = **4.32 seconds per day**. One week: about 30 seconds. One month: about 130 seconds. One year: about 1,580 seconds, or **26 minutes**. (a) At `Ts` = 1 hour, a year of drift is 26 minutes, which is about 44 per cent of the interval – **not acceptable**, and this is the surprise in the exercise, because an hour sounds forgiving. (b) At `Ts` = 1 second, one *day* of drift is 4.3 seconds, which is four

samples – unacceptable within a day. (c) Two gateways can drift in opposite directions, so the relative error is up to double, and correlation between them is meaningless within hours. All three cases say the same thing: synchronise, and verify that the synchronisation is actually running.

9.13.7. Age of twin has risen while completeness stayed at 100 per cent, so **the data is all arriving – it is arriving late.** Three causes: (i) the source has slowed, so event times themselves lag; (ii) your ingest has a growing backlog, so tuples are written long after their event time; (iii) a clock has drifted, so event times are being written into the past. **The measurement that distinguishes them: ingest time minus event time.** If it is growing, the delay is downstream of the source (ii) or the clock is drifting (iii); if it is unchanged while age of twin grows, the source is producing older readings (i). Distinguishing (ii) from (iii) is then one query: is the backlog visible as a queue depth?

9.13.8. *Hint plus the two that matter.* `good` – normal poll; use with normal gain. `uncertain` – reading within a few units of the sensor's range limit, or the sensor is past its recalibration date; **use with reduced gain**, per Sec. 9.6.2. `substituted` – somebody filled a gap; **do not correct toward it at all**, because correcting toward your own prior output makes the filter confident for no reason, which is the most damaging single answer in this exercise. `out_of_range` – a reading of 1023 or 0; ignore the value, count the event, and alert on the sensor. `no_reading` – the HTTP call failed; predict forward without a correction. `not_attempted` – the connector was stopped for a deployment; identical estimator behaviour to `no_reading`, different operational meaning.

9.13.9. Objection: "forward-filling does not remove missing data, it makes missing data indistinguishable from measured data – and the code that 'never has to handle' it is now silently averaging, fitting and validating against numbers the sensor never produced. Handling `no_reading` explicitly is a few lines in each consumer; recovering from a store full of unmarked substitutions is not possible at all." **The specific number: Chapter 7's hold-out spread of 2.0 reading units**, and everything derived from it – the 9-unit alert threshold, the 16 per cent detection floor, and the prediction range in Sec. 7.6.3. A forward-filled reading is closer to the model's prediction than a real one would be, so the spread is biased *low*, and the credibility argument overstates the twin.

9.13.10. No solution. Two predictions. The API's timestamp will turn out to be database-write time, and the way you will find out is by comparing two quantities polled in the same request and noticing they share a timestamp they cannot both have been read at. And your completeness will not be 100 per cent, because something will restart during the week – which is the right result, since a connector that has never seen a gap has never been tested.

9.14 Where to go next

In this book. Part III continues with the components this chapter hands off to. **Chapter 10 is the immediate sequel** and takes everything this chapter said to record – the tuple, the two timestamps, the quality state, the completeness counts, the watermark – and designs where it lives, how context joins to it, and how provenance is kept; the "Chapter 10's problem, created here" notes in Sec. 9.9.2 are literal. Chapter 11 builds the services that consume the streams, including the alerting that Sec. 9.7's detectors feed. Chapter

12 selects the platforms and tools this chapter deliberately named without recommending, and owns the edge-versus- cloud placement of Sec. 9.8.2. Chapter 13 covers the standards, which for this chapter means OPC UA's information model and the interoperability work around it [14], [15]. Chapter 14 owns the operational life of a connector: the recalibration schedules Sec. 9.2.2's drift figure implies, and the day a vendor changes a unit. Chapter 15 asks what happens when 400 substations become 400 twins.

In the literature, if you want more.

- *The one to read first:* [3] is a book chapter on sensing and communication for twins specifically, and covers properly what Sec. 9.3.2 declined to derive – sampling and aliasing, quantisation, medium access, compression – as well as time-series storage and soft-sensing.
- *Middleware and transports:* [1] is a systematic mapping of middleware for twins and is the source of this chapter's isolate-the-protocol framing; [4] is a survey and taxonomy of publish/subscribe systems, and is usefully honest that they cannot be compared on common ground; [2] treats connectivity as a platform framework.
- *Time:* [5] studies time discrepancy between digital and physical twins as a problem in its own right, which is what Sec. 9.5 is measuring.
- *Metrics:* [6] treats twinning rate and age of twin as measurable platform properties.
- *Data quality:* [7] on evaluating quality for applications with no human in the loop, which is the demonstrator's situation exactly.
- *When volume forces the architecture:* [8] and [9] on edge-cloud structural monitoring, which is Sec. 9.8.1's turbine arithmetic turned into a deployed system.
- *Consulted, not drawn on above:* [11] for the layer-by-layer threat classification behind Sec. 9.8.4, [10] on twins paired with constrained devices, [16] on communication interfaces as a named twin component, and [17] on what open-source twin frameworks provide at this layer.

In the demonstrator. Exercise 9.13.10 is the assignment and it is the first one in this book you can complete without writing a model: point a poller at the real API and run it for a week. Then do the thing that turns it into a connector rather than a script – add the six quality states, the completeness count, and the age-of-twin metric – and watch which of them fires first. It will not be the one you expect.

9.15 References

[1] A. Almeida, T. Batista, E. Cavalcante, F. Delicato, R. Motta, and M. Vieira, "Middleware for Digital Twins: A Systematic Mapping Study," in *Proceedings of the 1st International Workshop on Middleware for Digital Twin*, in Midd4DT '23. New York, NY, USA: Association for Computing Machinery, Dec. 2023, pp. 19–24. doi: 10.1145/3631319.3632302.

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

[3] C. Gomes, D. E. L. Rötter, A. Iosifidis, H. Feng, H. Ejersbo, and M. Frasheri, "Sensing and Communication of Data from the Physical Twin," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds., Cham: Springer International Publishing, 2024, pp. 147–171. doi: 10.1007/978-3-031-66719-0_7.

[4] A. Lazidis, K. Tsakos, and E. Petrakis, "Publish-Subscribe approaches for the IoT and the cloud: Functional and performance evaluation of open-source systems," *Internet of Things*, vol. 19, p. 100538, May 2022, doi: 10.1016/j.iot.2022.100538.

[5] M. Frasheri *et al.*, "Addressing time discrepancy between digital and physical twins," *Robotics and Autonomous Systems*, vol. 161, p. 104347, Mar. 2023, doi: 10.1016/j.robot.2022.104347.

[6] 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.

[7] K. Fizza *et al.*, "QoE in IoT: A vision, survey and future directions," *Discover Internet of Things*, vol. 1, no. 1, p. 4, Dec. 2021, doi: 10.1007/s43926-021-00006-7.

[8] 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.

[9] 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.

[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] 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.

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

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

[14] 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.

[15] 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.

[16] C. Steinmetz, G. N. Schroeder, R. N. Rodrigues, A. Rettberg, and C. E. Pereira, "Key-Components for Digital Twin Modeling With Granularity: Use Case Car-as-a-Service," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 1, pp. 23–33, Jan. 2022, doi: 10.1109/TETC.2021.3131532.

[17] S. Gil, P. H. Mikkelsen, C. Gomes, and P. G. Larsen, "Survey on open-source digital twin frameworks–A case study approach," *Software: Practice and Experience*, vol. 54, no. 6, pp. 929–960, Jun. 2024, doi: 10.1002/spe.3305.