Chapter 5: Data for Digital Twins

Before You Start

What you are assumed to know. You write software for a living, you are comfortable with HTTP APIs, JSON, a time-series or relational database, and reading Python you did not write, and you have read the previous four chapters. From Chapter 2 you have the plant rig's four low-level protocols – the I2C chain of sensors behind a multiplexer, the MODBUS bus that reads registers and writes coils, the InfluxDB point of measurement, tags, fields and timestamp, and the REST API on port 8099 whose sensing endpoints take a `since_timestamp` parameter. From Chapter 3 you have the two-question test and the staleness row of the shadow-versus-twin table: a shadow shows an old number, a twin acts on one and needs a bound. From Chapter 4 you have the seven building blocks, in particular the **data** block and the sentence that data is not state; the **physical adapter** that speaks the hardware's protocols; the shadowing pipeline whose first step is *normalise*; and the `as_of` field that says how old the state's belief is. Each is recalled in a clause where it is used.

What you are not assumed to know. Signal processing beyond arithmetic: the one theorem this chapter needs is stated in a single line, and the mathematics appendix at the end of the book has the argument. Any streaming, serialisation or time-series product: a few are named, only where a published system used them, and none is recommended. Any industrial standard: two protocols are introduced for what they do, and the standards chapter takes up the rest.

What this chapter covers. Chapter 4 named data as one of seven blocks and said that data is not state. This chapter is about that block: how a number gets from a sensor into the store the twin reads, and what is lost, gained and decided on the way. It reads the twinning loop as data and names the tuple the physical adapter produces; it sorts twin data three ways; it derives a sampling interval from a decision and shows what goes wrong when the interval is chosen from the datasheet instead; it reads a sensor's datasheet for the six numbers that change a design; it puts four timestamps on one path and two clocks behind them; it asks five things of any transport, names the protocols and shows one stream of readings as each carries it; it encodes one reading six ways and counts the bytes; it sorts a twin's functions by what they do and builds the ingestion pipeline as five of them; it places stream processing on a continuum from edge to cloud; it lays the same readings out as rows, columns, a time-series point, an object and a metadata record, and sizes the volume; it gives quality its own field; it ends with a choosing table, the rig's own data path, and three faded examples.

What this chapter deliberately does not cover. The downward leg of the loop – commands to the physical twin and the command path that guards them – beyond noting where it joins. State estimation, and every check that needs the model to say what a reading *should* have been, including the stuck sensor: those belong to the state and estimator chapter. How a model is built or kept current: the models chapter. Whether the twin deserves to be believed: the credibility chapter, which will inherit this chapter's limitations paragraph.

Learning objectives. By the end of this chapter you will be able to:

1. **Read** a sensor's datasheet for the six numbers that change a design, and **decide** whether a measurement can support a stated decision.

2. **Derive** a sampling interval from a decision, and **design** the aliasing check that tests it.
3. **Evaluate** any transport against the five properties a twin needs, and **place** HTTP, Message Queuing Telemetry Transport (MQTT), Open Platform Communications Unified Architecture (OPC UA), MODBUS and Kafka against them.
4. **Carry** event time and ingest time through a pipeline, **size** clock error against the sampling interval, and **compute** twinning rate and age of twin.
5. **Choose** an encoding for a value in flight and a store and layout for data at rest, and **say** what each preserves and loses.
6. **Sort** a twin's functions by what they do, **write** the functions of an ingestion pipeline as testable, versioned units, and **place** each on the adapter or the core.
7. **Design** a time-series schema, a retention policy, a cold-path layout and a metadata record, and **size** the volume before it is a problem.

Every section that follows serves at least one of those seven, and the exercises at the end say which.

The Twinning Loop, Read as Data

Chapter 3 drew the twin as a control loop: readings go up from the physical twin, commands come down to it. Chapter 4 opened the digital object and put a state at its centre. This chapter is the upward leg, from the sensor to the store, and it ends where Chapter 4's state begins. Figure 1 draws the loop with the upward leg labelled by the stations this chapter visits.

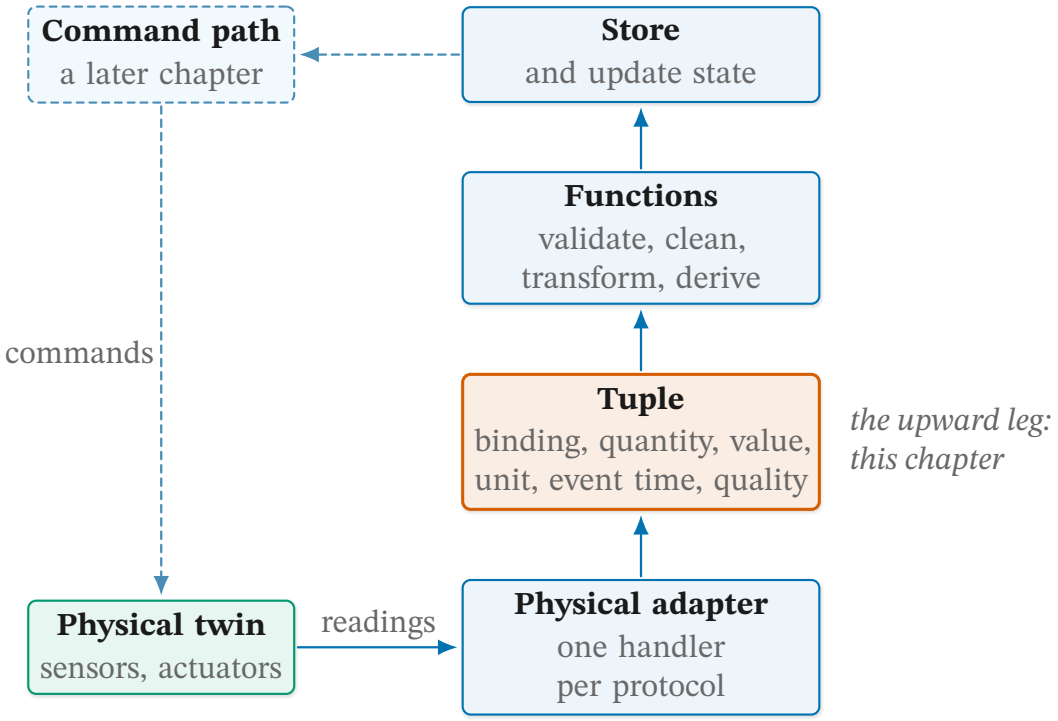


Figure 1: The twinning loop read as data. Readings climb the right-hand side from the physical twin through the adapter's tuple and the functions into the store and the state; commands come back down the left through a command path a later chapter owns. This chapter is the upward leg.

Take an offshore wind turbine as the generic instance for this section and most of the chapter. Its nacelle carries a few hundred sensors – bearing temperatures, generator speed, tower accelerations, wind speed and direction – wired into a controller that the turbine's maker designed for the turbine's own control, not for you. A twin of that turbine begins where a reading leaves that controller.

The physical adapter of Chapter 4 speaks whatever the hardware speaks. What it produces, whatever it spoke, is one thing: a **tuple** of six fields – binding, quantity, value, unit, event time and quality – and the Transports section fixes that tuple as the adapter's output contract once the transports that feed it have been named. Until then, hold one sentence: the adapter's whole output is a stream of tuples, and every section of this chapter is about one of the six fields or about the stream they form.

Why the adapter is a quarter of the engineering

In a project plan the adapter is one line, estimated as "read the API", and in the delivered system it is a quarter of the engineering. Three reasons, each of which the rest of the chapter unpacks.

The physical twin was not designed for you. The turbine's controller exposes what its maker needed for control, in the units, rates and naming its maker chose, over the protocol its maker's suppliers used. A register holding the bearing temperature in tenths of a degree, as a sixteen-bit integer, big-endian, refreshed every second, is a normal thing to be handed. Every one of those facts has to be discovered, written down as a binding, and applied exactly once. The middleware literature treats this heterogeneity as the problem middleware exists to absorb: twins, like the Internet of Things (IoT) before them, encompass heterogeneous devices, software components, communication protocols and legacy applications, and middleware's traditional job is to hide that behind standardised interfaces [1]. In a twin that job lands on the adapter.

Every failure in the physical world arrives through this component. A sensor fails, a cable is cut, a gateway reboots with the wrong time, a firmware update changes a scale factor. None of those events announces itself. Each arrives at the adapter as a tuple that looks wrong, or as the absence of a tuple that was due. The adapter is the only component positioned to notice, and the Quality section is about giving it the vocabulary to say what it noticed.

Every number the credibility argument will rest on is a tuple it produced. When the credibility chapter asks whether the twin's model is good enough to act on, the evidence is a comparison between what the model predicted and what the sensors reported. If the sensors' readings were mis-scaled, mis-timed or silently filled in, the comparison is meaningless and nothing downstream can tell. A large part of whether a twin deserves to be believed is settled in the adapter, before any model runs.

A Vocabulary for Twin Data

Chapter 4 sorted the data block into incoming, recorded and configuration. That sorting is by where data sits. This section adds two more – by where it came from and by how quickly it must be used – and a third that runs across an asset's life. Table 1 puts the three side by side.

By origin

Sensed data came from a sensor through the adapter: a tuple with a binding to something physical. **Simulated** data came from a model run by a simulator: the same shape of tuple, with a binding to a model version rather than to a sensor. **Derived** data was computed by a function from sensed or simulated data: a residual, a rolling mean, an efficiency. **Configuration** data is what the twin was told about itself and its asset: bindings, calibration constants, thresholds, firmware versions.

Derived data is the category people forget to store, on the argument that it can be recomputed. It cannot, once the function that derived it has changed. A residual computed by last year's threshold is a different number from one computed by this year's, and if only the sensed data was kept, the record of what the twin believed at the time – and therefore of what it did, and why – is gone. Store derived data, and store beside it the version of the function that made it. The Functions section returns to the versioning; here the point is the storing.

By timeliness

Assign every service to one of three paths by how quickly it needs its data. The **hot path** is for answers within seconds or less – an alarm on a vibration threshold, a command-path check. The **warm path** is for answers within minutes – a residual recomputed per check interval, a dashboard that refreshes. The **cold path** is for answers within hours to months – a fatigue summary, a recalibration, a report. The names come from the data-analytics reference architectures that the manufacturing-twin literature reviews, one of which defines three paths for data after ingestion: a hot path for near-real-time insight through stream analytics, a warm path for more detailed processing at larger delay, and a cold path for batch processing at longer intervals [2].

Assigning each service to a path fixes three other decisions at once – the protocol that carries its data, where it runs, and which store it reads – which is why the path is the first thing to decide and the last to change. A hot-path alarm cannot wait for a nightly file; a cold-path report should not be reading a message broker. The How to Choose section is built on this observation, and the table there is organised by path.

By lifecycle

An asset has a life – design, manufacture, operation, retirement – and data about it is produced in every phase by different tools that never meet. The **digital thread** is the name for the linkage of one asset's data across those phases. In the Digital Twin Consortium's definition it is the means by which information about an asset is tied together along several axes of its virtual representation – time and lifecycle stage among them, with the design intent counted as a stage [3]; an accessible overview puts the same idea as a twin being both a shadow of its physical twin's present state and a thread recording its evolution over time, which applications can scan and draw meaning from [4]. For a software engineer the thread is a habit more than a technology: record, on every row the twin stores, which firmware produced the reading and which model version produced the prediction, so that a number from March can still be interpreted in November after both have changed.

Table 1: Three sortings of twin data. Chapter 4's incoming, recorded and configuration sorting is by where data sits; these three are by where it came from, how fast it is needed, and which phase of the asset's life produced it.

Sorting	Classes	What it decides	Turbine instance
By origin	sensed, simulated, derived, configuration	what must be stored, and with which version	a bearing temperature; a predicted temperature; their residual; the bearing's alarm threshold
By timeliness	hot, warm, cold	protocol, placement, store	vibration alarm; per-check residual; monthly fatigue summary
By lifecycle	design, manufacture, operation, retirement	which firmware and model version a row was produced under	the blade's design loads; its factory inspection; its operating history; its decommissioning record

Read Table 1 across its last column: one bearing temperature appears in all three rows, as a sensed reading, as a hot-path alarm input and as a row stamped with a firmware version, and the three sortings are three questions asked of the same tuple rather than three kinds of data. None of the three is state: the Functions section says what turns data into state, and why nothing in this chapter does.

Sampling: From Physical Quantity to Number

A sensor reports a continuous quantity; the twin stores a sequence of numbers. Between the two sit two decisions that are usually made by default and should be made on purpose: how often to read, and how finely to round. This section makes both, and then shows the failure that the default reading interval produces.

Five steps from a decision to an interval

The sampling interval is derived from the decision, not chosen from the datasheet. Five steps:

1. **Name the decision and how long it may take.** For the turbine: "take the turbine out of service before a bearing fails", and the decision may take a day, because a vessel has to be scheduled.
2. **Name the physical event that matters to the decision, and how long it takes to become visible in the quantity you measure.** A bearing that is failing runs hot over days to weeks before it fails; its temperature climbs over hours.
3. **Sample several times across that event.** If the temperature climbs over hours, a reading every ten minutes sees the climb as a curve rather than as one surprising

number. Several means at least a handful; the appendix says how many, and why.
4. **Check the ceiling.** The sensor's own response time and maximum sample rate, the bus that carries the reading, and the power budget of a battery-fed gateway all set a rate above which more readings cost more and tell no more.
5. **Check the floor.** The decision's latency sets a rate below which the twin learns too late. A day's decision latency with a ten-minute reading is comfortable; a ten-minute reading against a decision that must be made within a minute is not.

Write the interval as T_{s}, and the rate as $f_{\mathrm{s}} = 1/T_{\mathrm{s}}$. For the bearing, $T_{\mathrm{s}} = 600$ seconds. Now change the decision. "Shut the turbine down within a second if the tower's acceleration exceeds its design load" names a physical event – a gust, a resonance – that becomes visible in milliseconds, and step three then asks for readings a few milliseconds apart: T_{s} in the hundreds of microseconds, f_{s} in the kilohertz. Same turbine, two quantities, two decisions, intervals four orders of magnitude apart. The interval belongs to the decision, and a twin that reads everything at one rate has made one of the two decisions wrong.

Aliasing: fast behaviour appearing as slow behaviour

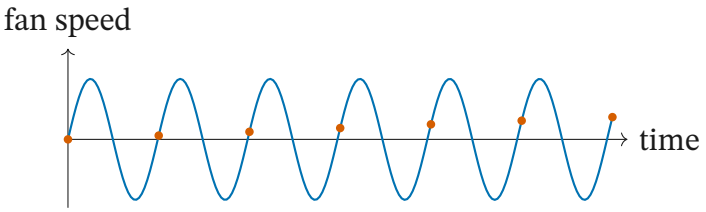
The failure that step three guards against has a name. **Aliasing** is fast behaviour appearing as slow behaviour when the quantity is sampled more slowly than it changes. The record then contains a rhythm that does not exist, and nothing in the record says so.

One worked instance. A building's ventilation fan cycles on and off on a timer, one full cycle every 594 seconds – a "ten-minute" cycle that is in fact nine minutes and fifty-four seconds. The room's temperature follows it with a small ripple. The building twin samples the room temperature every 600 seconds, because ten minutes seemed reasonable for a room. Each sample lands six seconds later in the fan's cycle than the one before, so the samples walk slowly round the cycle, and the record shows a smooth, slow oscillation whose period is the time it takes six seconds per sample to add up to one full cycle:

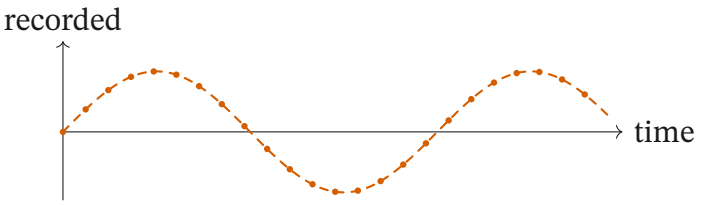
$$f_{\mathrm{alias}} = \frac{1}{594} - \frac{1}{600} = \frac{6}{356{,}400} = \frac{1}{59{,}400}$$

per second, a period of 59,400 seconds – sixteen and a half hours. Figure 2 draws it. Panel (a) is one hour of the true signal with the six samples marked on it; panel (b) is a day of the samples alone, a clean phantom rhythm that a colleague will explain as "the building warming through the afternoon and cooling at night" and a model will be fitted to.

The empirical check costs an afternoon and is worth it for every quantity a twin samples. Sample the quantity much faster than you intend to, once – every second, for a day. Then **decimate**: keep every six-hundredth reading, which is what your intended interval would have seen. Compare the decimated record with the fast one. If the decimated record contains a rhythm the fast one does not, you have found an alias, and the fix is to sample faster, or to average the fast readings over each interval rather than pick one, or to move the sampling instant so that it does not beat against a known cycle. The rule the check is testing is the one theorem this chapter states, and it fits on a line: a sampled record represents the quantity faithfully only when the sampling rate exceeds twice the fastest rate at which the quantity changes, $f_{\mathrm{s}} > 2 f_{\max}$. The mathematics appendix has the argument and the reason "several times across the event" in step three is the practical form of it.



(a) a 594 s cycle sampled every 600 s, one hour



(b) the same samples over a day: a phantom 16.5 h rhythm

Figure 2: Aliasing. In (a) a room temperature that follows a fan's 594-second cycle is sampled every 600 seconds; in (b) the samples alone, over a day, draw a slow wave with a period of sixteen and a half hours that the room never had. Nothing in the record of (b) says it is a phantom.

Quantisation: the ten-bit calculation

The second decision is how finely to round. An analogue-to-digital converter reports one of a finite set of counts, and the **quantisation step** q is the size of one count in the quantity's own unit. Work it for a ten-bit converter reading a temperature sensor that outputs ten millivolts per degree, powered from a 3.3 V reference. Ten bits give $2^{10} = 1024$ counts across the reference, so one count is

$$q = \frac{3.3}{1024} = 0.00322$$

volts. At ten millivolts per degree that is $0.00322/0.010 = 0.322$ degrees per count. The converter rounds to the nearest count, so the worst-case rounding error is half a count:

$$e_q = q/2 = 0.16 \tag{1}$$

degrees. Equation 1 is the floor under every later use of this reading: a residual of a tenth of a degree from this sensor is inside the rounding and means nothing, and a compression that promises to reconstruct the value within a tenth of a degree is promising more precision than the reading ever had. The Storage section uses that second fact. Note also which number the calculation did not use: the sensor's accuracy, which the Sensors section defines. Quantisation is the converter's error; accuracy is the sensor's; the two add, and a reading's honest uncertainty is never smaller than the larger of them.

Sampled versus event-driven acquisition

Everything so far has been **sampled acquisition**: read the quantity at a fixed interval, whether or not it changed. Its alternative is **event-driven acquisition**: emit a tuple when

something happens – a breaker trips, a door opens, a dose is delivered, a value crosses a threshold. Events carry information that sampling misses (the breaker tripped at 06:10:14, not "sometime in the ten minutes before 06:20") and they carry nothing when nothing happens, which is efficient and dangerous. Silence from an event-driven source means either that nothing happened or that the source is dead, and the two are indistinguishable from the record. The resolution is a **heartbeat**: a periodic tuple that says "nothing happened, and I am alive". With it, silence has a meaning; without it, a dead sensor and a quiet day are the same record.

The sentence to carry: sample the state, capture the events, align them on time. Quantities that describe a condition – temperatures, pressures, levels – are sampled. Things that happen – trips, doses, commands, alarms – are captured as events. And both are placed on one timeline by their event time, which is the Time section's subject. First, though, the sensor that produced the sample, and what its maker actually promised about it.

Sensors: What You Are Actually Given

A sensor is a model of the world that somebody else built and documented. A capacitive soil probe measures capacitance; the datasheet claims a relationship between that capacitance and the water content of the soil around it, and the claim holds under the conditions the datasheet states and nowhere else. A bearing's temperature sensor measures the temperature of a small metal bead pressed against the bearing housing, and the datasheet claims a relationship between that bead and the bearing. Everything the twin will ever know about the physical twin arrives through such claims, and reading them is the skill this section teaches.

Six numbers

A datasheet is long. Six numbers on it change a design, and Table 2 names what each one changes. Two of them, resolution and maximum sample rate, the Sampling section has already leaned on; the other four are new here.

Table 2: The six datasheet numbers that change a design, and what each changes. Range and resolution are about what can be reported; accuracy and drift about how far to trust it; response time and sample rate about how fast it can be asked.

Number	What it says	What it changes downstream
Range	the smallest and largest values the sensor reports	the out-of-range quality state; whether a fault can be seen at all
Resolution	the smallest change the sensor can report	the quantisation step; how many digits to store; the floor on any threshold
Accuracy	how far a report may be from the truth, under stated conditions	how small a residual can mean anything; the error bound a compression may use

Number	What it says	What it changes downstream
Drift	how accuracy degrades with time or temperature	the recalibration interval; whether a slow trend is the asset or the sensor
Response time	how long the report takes to follow a change in the quantity	the ceiling on useful sampling rate; the lag the model must expect
Maximum sample rate	how often the sensor can be read	the ceiling on the sampling interval; the bus load

A published fermentation twin shows what the numbers look like when they are written down: its sampling chamber's sensor array has an optical dissolved-oxygen sensor with a range of 0 to 100 mg/L and an accuracy of plus or minus 0.05 mg/L, and an industrial pH sensor with a range of 0 to 14 and an accuracy of plus or minus 0.002 pH with temperature compensation, beside conductivity, temperature and pressure sensors and an electronic hydrometer for density [5]. Read those two accuracies against the decisions they serve. A brewer deciding whether a fermentation has stalled needs to see a change of a few tenths of a pH unit over hours; an accuracy of two thousandths is a hundred times better than the decision needs, and the design question becomes whether the sensor's drift over a three-week fermentation stays inside that margin. The same paper notes that the manual practice it replaces was a master brewer drawing samples by hand for gravity, dissolved oxygen, pH and conductivity – which is the decision the sensors were bought to improve, and the number to read the datasheet against.

Calibration is the number the datasheet cannot give, because it is a date rather than a figure, and it is the cost most often skipped. It does not stay skipped: a twin fed with real-time sensor data is fed with data affected by environmental factors, wear and calibration issues, which show up as data loss, noise and inconsistencies [6]. A sensor whose calibration has drifted reports a trend that the asset does not have, and the Quality section's `uncertain` state is the only honest thing the adapter can attach to it until somebody recalibrates.

Resolution is not accuracy

Resolution and accuracy are the most commonly confused pair on the table, and the confusion runs one way. Resolution is how finely a sensor can *report*; accuracy is how far the report may be from the *truth*. A reading of 21.437 degrees from a sensor accurate to half a degree has four digits, two of which are decoration. Nothing in the tuple says so. The value arrives with three decimal places because the analogue-to- digital converter had that many bits, and a dashboard shows them because they are there, and a colleague reads "21.437" as a claim about the world when the honest claim is "somewhere between 20.9 and 21.9".

Two consequences for the twin. Store the value at the sensor's resolution and no finer – extra digits cost storage and invite false precision. And never set a threshold, an alarm or a residual test finer than the accuracy: a rule that fires when the residual exceeds a tenth of a degree, on a sensor accurate to half a degree, fires on the sensor and not on the asset.

Time: Four Timestamps and Two Clocks

A tuple carries an event time. Where that time came from, how it is written, and how far it may be trusted decide whether two readings can be compared at all. This section sits before the Transports section because the five properties a twin asks of a transport presuppose that event time exists and is preserved.

Four stations on one path

Follow one bearing temperature from the sensor to the twin's store, and count the places a clock is read. Figure 3 draws them.

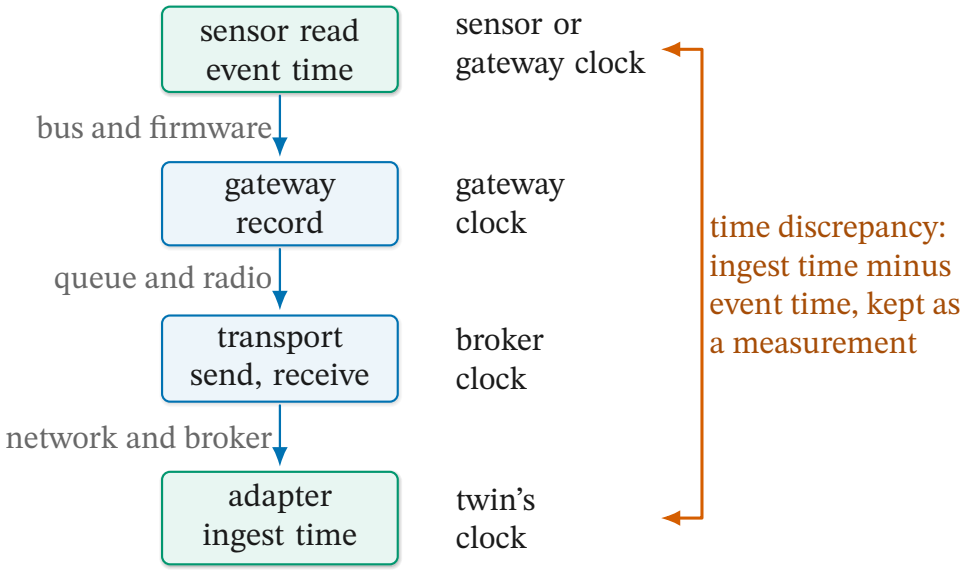


Figure 3: Four stations on one reading's path, each with a clock and each gap with a different cause and owner. The tuple keeps the first and the adapter records the last; their difference is the time discrepancy.

The **sensor read** is when the quantity was measured, by the sensor's own clock if it has one, or by the gateway's when it polled. The **gateway record** is when the gateway wrote the reading into its local log or message queue; between the first and the second lie the bus and the gateway's firmware. The **transport** stamps are when the message was sent and when it was received by a broker or server; between the second and the third lie a queue, a radio and a network. The **ingest time** is when the twin's adapter turned the message into a tuple and wrote it; between the third and the fourth lie the broker and the twin's own load. Each gap has a different cause, and each has a different owner: the sensor's maker, the gateway's integrator, the network's operator, the twin's developer. When a reading arrives late, the first question is which gap grew, and a path that recorded all four stamps can answer it. A path that recorded only the last cannot.

Event time, ingest time, and the difference between them

Of the four, two must survive into the tuple and the store. The rule: the tuple carries the **event time** t_e, the adapter records the **ingest time** t_i, and their difference is not an error to be hidden but a measurement to be kept, because it is the one number that says how far behind the physical twin the digital one is running. Write it as

$$D = t_{\mathrm{i}} - t_{\mathrm{e}} \tag{2}$$

and call it the **time discrepancy**. Equation 2 has been studied in its own right for twins: the coupling between a twin and its physical counterpart is subject to time delays and synchronisation errors, timeliness can be critical when a twin must warn a physical system of an obstacle or when delay threatens a control loop's stability, and one line of work brokers external data into a running co-simulation in real time precisely to manage that delay [7]. It is also an incident indicator: a study that subjected a twin to denial of service, packet loss with jitter, replay, out-of-order delivery and node failure found that each left measurable deviations in the twin's operational metrics – timeliness, reliability, availability and consistency – and proposes metric-based observability as a lightweight way to detect that something is wrong [8]. A time discrepancy that grows is the first of those metrics to move, and it costs one subtraction.

The failure the rule prevents is the adapter stamping each tuple with the time it *arrived* and calling that the event time. On a quiet network the two are seconds apart and nobody notices. When the gateway buffers an hour of readings through a link outage and replays them, every reading in that hour carries the same minute, the record shows the bearing's temperature leaping and freezing, and the estimator of Chapter 7 fits a model to the network outage.

The four timestamp questions

Every timestamp a twin stores should be able to answer four questions.

Whose clock? The sensor's, the gateway's, the broker's or the twin's. Two timestamps from two clocks can be compared only if the clocks agree, and the next subsection is about how much they must.

What offset? A time written without its zone is a time written in somebody's local convention. Store every timestamp in Coordinated Universal Time (UTC), and let the dashboard convert. A twin that stores local time loses an hour every autumn and gains one every spring, and the gap and the duplicate it produces are indistinguishable from a sensor fault.

What resolution? A timestamp to the second cannot order two readings taken a millisecond apart. Store event times at the resolution the sampling interval needs – milliseconds for a kilohertz stream, seconds for a ten-minute one – and no coarser.

Start or end of interval? A ten-minute average stamped 06:10 either describes 06:00 to 06:10 or 06:10 to 06:20, and the two conventions are both in use. Decide once, write it in the binding, and never let a downstream function guess.

Clock error as a fraction of the sampling interval

Clock error matters as a fraction of the sampling interval. Write the error between two clocks as ε and the sampling interval as T_{s}; the quantity to look at is $\varepsilon / T_{\mathrm{s}}$. Work it twice, with opposite answers.

A gateway clock one second wrong, against a bearing temperature read every ten minutes: $\varepsilon / T_{\mathrm{s}} = 1/600$, under two tenths of a per cent of the interval. The reading is placed a

second from where it belongs on a curve that changes over hours. Noise. Network time synchronisation, which holds a clock within tens of milliseconds over an ordinary network, is more than this reading needs.

The same one second, against a tower acceleration sampled at two kilohertz: $T_s = 0.0005$ seconds, so $\varepsilon / T_s = 2000$. Two thousand samples are in the wrong place. An analysis that compares two accelerometers on the tower to find the direction of a vibration is comparing samples a second apart and calling them simultaneous, and the direction it finds is fiction. This stream needs a clock disciplined to microseconds – precision time protocols over a wired network, or a satellite time reference at the gateway – and the decision to buy that is made here, from this ratio, not from the sensor's datasheet.

One special case of clock error is not a fraction at all. A gateway that reboots without a battery-backed clock starts at some epoch – the year 1970, or the firmware's build date – and stamps every reading with an impossible time until it synchronises. Those tuples must be rejected, not corrected: a reading stamped 1970 cannot be placed, and guessing where it belongs is inventing data. The validate function of the Functions section owns that rejection, and the Quality section gives it a state to attach.

Twinning rate, age of twin, time discrepancy

Three numbers, computed per binding, say how well the upward leg is working. The **twinning rate** R is how often the twin's state is updated from the physical twin – distinct from the sensor's sampling rate, which may be higher, with the gateway averaging, or lower, with the adapter interpolating. The manufacturing literature defines it as the synchronisation rate between physical and virtual entity and notes that most published twins did not distinguish it from the data sampling frequency [9]; the platform literature normalises it to an interval where zero means no synchronisation and one means real-time, and observes that a low rate leads to an outdated state [10]. Compute it as updates per unit time for each binding, $R = n / T$, and alarm when it falls.

The **age of twin** A is the time that has elapsed since the newest reading the state rests on:

$$A = t_{\text{now}} - t_{\text{e}} \tag{3}$$

where t_{e} is the event time of that newest reading. The platform survey defines age of twin as the time elapsed since the twin last received an update from its physical counterpart, the temporal freshness of the twin [10]; Equation 3 uses event time rather than arrival time so that a replayed backlog does not make a stale twin look fresh. This is the best single health number a twin has. It says, in one figure, whether the twin is shadowing at all (A grows without bound when the link is down), whether it is keeping up (A stays near T_s when it is), and whether a command may be issued (Chapter 3's staleness bound is a bound on A). It is what Chapter 4's `as_of` field exists to make computable: `as_of` is t_{e} for the state, and A is now minus that. Every state-centric service should show it, and the lifecycle field should fall to *out of sync* when it exceeds a bound the configuration names.

The **time discrepancy** D, from Equation 2, is the third, and the one whose growth points at a gap on the path rather than at the asset. Together the three answer "is the twin current?", "how current?" and "if not, whose fault?" – and all three are computable from the tuple's event time and the adapter's clock, which is the whole argument for carrying both.

Moving the Data: What a Twin Needs from a Transport

A reading exists first as a voltage, then as a register, then as a message, then as a tuple. This section is about the middle two steps: how a value travels, and what a twin must be able to ask of whatever carries it.

Two layers and a gateway

Transports come in two layers, and Figure 4 puts a gateway between them.

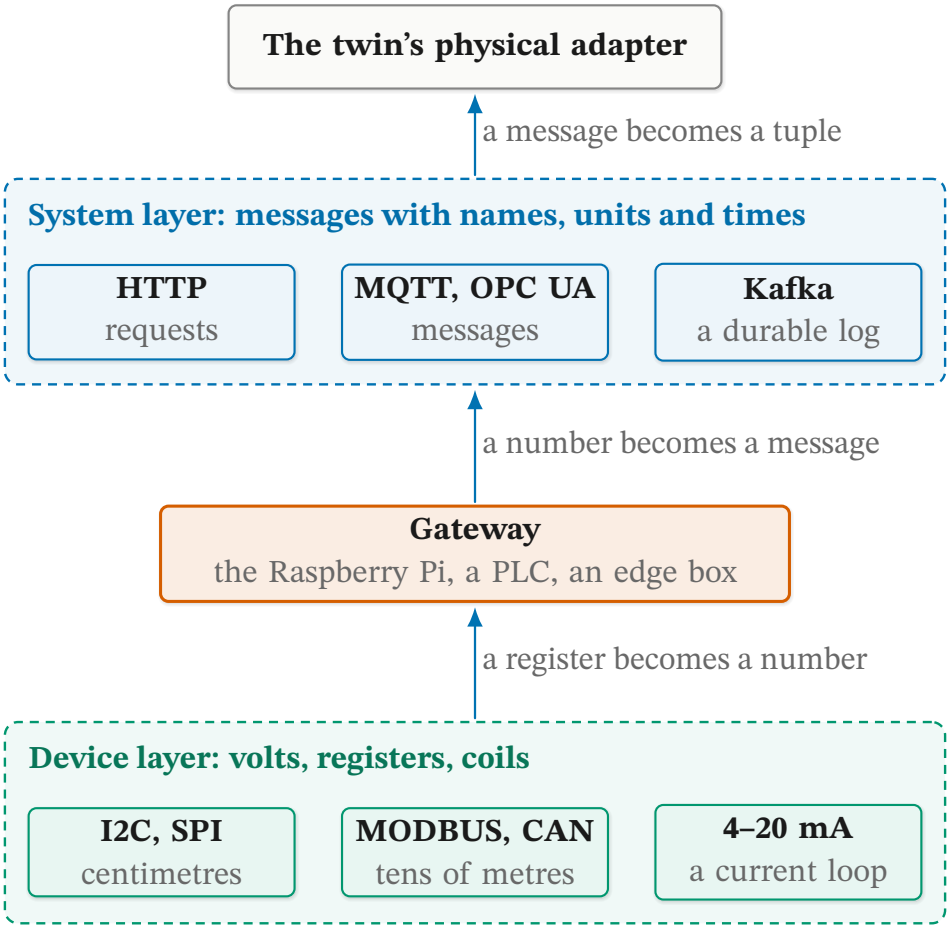


Figure 4: Two layers of transport. The device layer carries volts and registers over short buses; the system layer carries messages with names, units and times; the gateway is where one becomes the other, and the twin's adapter reads the system layer.

The **device layer** is the buses Chapter 2 introduced and a few more of their kind: the Inter-Integrated Circuit bus (I2C) and the Serial Peripheral Interface (SPI) over centimetres on a board; MODBUS over RS-485 and the Controller Area Network (CAN) over tens of metres in a cabinet or a vehicle; the 4–20 mA current loop, in which a sensor signals a value as a current between four and twenty milliamps down a two-wire pair, so that zero current means a broken wire rather than a zero reading. These carry volts, registers and coils. They have no names for quantities, no units and no timestamps; what they carry is a number at an address.

The **system layer** is where the twin's software lives: messages over a network, each with a name, a unit if the sender attached one, and a time. HTTP, MQTT, OPC UA and a durable

log such as Kafka live here.

Between them sits the **gateway**: Chapter 2's Raspberry Pi, a programmable logic controller (PLC) in a cabinet, an industrial edge box. It polls the device layer, turns a register into a number, attaches a name and a time, and emits a message. A gateway is where a register becomes a number and a number becomes a message; the twin's adapter is where a message becomes a tuple. Which of those two does the scaling and the unit is a decision, and the section's last subsection makes it.

Poll versus subscribe, and the question that outranks both

Every system-layer transport is either **polled** – the twin asks, the source answers – or **subscribed** – the twin registers an interest and the source, or a broker on its behalf, pushes. Table 3 contrasts them.

Table 3: Poll versus subscribe. Neither is better; the row that decides between them for a twin is the last one, and the question below outranks both.

	Poll	Subscribe
Who initiates	the twin	the source or a broker
Timing of a reading	when the twin asks	when the source publishes
Cost when nothing changes	a request and an empty answer	nothing
Cost when much changes	readings missed between polls	a burst the twin must absorb
Connection state	known at every request	known only through keep-alives or a broker's notice
Fan-out to many consumers	each polls the source	the broker fans out
What a gap looks like	a poll that failed, visible to the twin	messages that never arrived, invisible unless numbered

Resumability – **can I ask you for what I missed** – is the question that outranks the choice between the two. A transport that can answer it turns a dropped connection into a delay; one that cannot turns it into a gap that nothing downstream can fill. A polled source with a history parameter – give me everything since this time – is resumable. A subscribed feed whose broker keeps a log and lets a subscriber resume from where it left off is resumable. A subscribed feed whose broker delivers each message once, to whoever is connected, and forgets it, is not, however low its latency. Ask this question of every transport before asking any other.

Five properties a twin needs

Chapter 3's obligations arrive at the transport as five properties, each of which has a victim when it is missing. Table 4 names them.

Table 4: Five properties a twin needs from any transport, and the component that breaks when each is missing. The five are the transport-side form of Chapter 3's staleness, fail-safe and concurrency obligations.

Property	The question	Who breaks without it
Event time preserved	does the message carry when the reading was taken, not when it was sent?	the estimator, which fits a model to arrival times
Connection state observable	can the twin tell a live source that is quiet from a dead one?	the monitoring service, for which a lost link looks like a quiet day
Gap survivable	after an outage, can the twin recover what it missed?	the record, which has holes nobody can see
Ordering knowable	can the twin tell whether two messages arrived in the order they happened?	the shadowing function, when a correction arrives before the reading it corrects
Deduplication possible	if a message is delivered twice, can the twin tell?	every aggregate, which counts a retried reading twice

Score a transport by counting the properties it gives you for free, then by counting the ones you can add above it – a sequence number in the payload adds ordering and deduplication to anything – and reject the ones where a property cannot be added at all. The scores are worked for the rig in the demonstrator section and for a turbine in the faded examples.

Protocols

With the questions settled, the protocols. One paragraph each on what a protocol is for, in the order a reading meets them from the device upward, and beside each a sample of the same data as that protocol carries it; the How to Choose section ranks them by path, not here. The data is a stream of bearing temperatures from the turbine, read every ten minutes, written here as JSON with its timestamps – the stream the Encodings section will re-encode and the Storage section will store:

```
{"binding": "wt07/gen-bearing/temp", "event_time":
 ↪  "2026-06-09T06:00:00Z",
 "value": 61.1, "unit": "degC", "quality": "good"}
{"binding": "wt07/gen-bearing/temp", "event_time":
 ↪  "2026-06-09T06:10:00Z",
 "value": 61.2, "unit": "degC", "quality": "good"}
{"binding": "wt07/gen-bearing/temp", "event_time":
 ↪  "2026-06-09T06:20:00Z",
 "value": 61.4, "unit": "degC", "quality": "good"}
```

One object per reading, shown here over two lines each, with the tuple's fields as keys; the quantity is folded into the binding's name, which is a convention the binding records. Each protocol below carries these readings differently, and what differs between them is exactly the five properties of Table 4.

HTTP is for polling, and Chapter 2 has already used it. It is a request/response protocol that addresses resources by a Uniform Resource Identifier (URI), text-based, with no fixed header or payload size [11]. Its resumability is entirely in the API's hands: a sensing endpoint that accepts a history parameter – readings since a given time – makes polling resumable and gives the twin ordering and deduplication through the timestamps it returns; one that returns only the latest value gives none of the five. The turbine's controller, if it offers such an endpoint, is asked for the stream like this:

```
GET /turbines/wt07/sensing/gen-bearing/temp?since=2026-06-09T05:50:00Z
Host: scada.windfarm.example
Accept: application/json
```

and answers with a JSON array of the readings since that time – the three objects above, inside `[` and `]`, newest first or oldest first as the API documents. The `since` parameter is the whole of the resumability; without it the same endpoint returns one object and the twin learns nothing about what it missed. HTTP's cost is the request itself, paid whether or not anything changed, and its virtue is that every request reveals the connection state.

MQTT is for brokered telemetry. It is a publish/subscribe transport standardised by the Organization for the Advancement of Structured Information Standards (OASIS) and by ISO, designed to minimise network bandwidth and device resource requirements: a publisher sends a message on a named topic, a broker relays it to every subscriber of that topic, and the fixed header is two bytes [11], [12]. Brokers differ in the features they support – three quality-of-service levels, a retain flag that keeps the last message on a topic for late subscribers, persistent sessions, and a last will and testament published by the broker when a client disappears [12]. One reading, published:

```
topic:    fleet/wt07/gen-bearing/temp
qos:      1         retain: false
payload:  {"seq": 4127, "event_time": "2026-06-09T06:10:00Z",
           "value": 61.2, "unit": "degC", "quality": "good"}
```

The binding has moved into the topic name, and one field has appeared that the stream did not have: `seq`. Bare MQTT gives a twin little of the five – no event time unless the payload carries one, connection state only through the will message, no gap recovery, no ordering or deduplication across a reconnect – and the industrial conventions layered on it add those back in the payload: a sequence number so a gap and a duplicate are visible, a timestamp so event time survives, a will message registered at connection so the broker announces a client that vanished. Read against the table: connection state, ordering and deduplication, added by convention on top of a transport that had none of them, and every one of those conventions is the publisher's promise rather than the protocol's.

MODBUS is the register-based poll/response protocol of legacy PLCs, recalled from Chapter 2 and belonging to the device layer. A security survey characterises it in a line: a legacy industrial protocol that works on request/reply messages and has no inherent encryption [13]; the predictive-maintenance review finds it among the protocols the twins it surveyed

used to reach equipment [14]. The same reading, as the gateway polls it from the turbine's controller – the request and reply with the bus's own address and checksum bytes left off:

```
request:  03  00 2A  00 01     read holding registers, address 42, count
 ↪  1
reply:    03  02  02 64        function 3, two bytes follow: 0x0264 = 612
```

Two bytes carry the value, and nothing carries anything else. That 612 is tenths of a degree Celsius, that register 42 is the generator-side bearing of turbine 7, that the reply arrived at 06:10:00 by the gateway's clock and that the reading is believed good – every field of the tuple but the value – exists only in the binding and in the gateway that applies it. So every one of the five properties has to be supplied by whoever polls it, and that is the gateway's job: poll, stamp with its own clock, name the register from the binding, and publish upstream with a sequence number.

OPC UA is the protocol that carries an information model as well as values. It is based on two communication models, client/server and publish/subscribe, and provides a semantically enriched information model to represent data and its meaning [15]; a client browses a server's address space to discover variables and the information attached to them [16]; generic device models can be instantiated for particular devices, and both a publish/subscribe pattern and a request/response pattern are supported on the same address space [9]. It standardises manufacturer-independent data exchange between physical objects in industrial environments [14] and stands as the reference standard for machine interoperability: the interoperability literature treats it as one of the main standards that other descriptions are mapped onto [15]. The same reading as an OPC UA data value, in the shape of its JSON encoding and simplified:

```
{"NodeId": "ns=2;s=WT07.GenBearing.Temp",
 "Value": {"Type": "Double", "Body": 61.2},
 "SourceTimestamp": "2026-06-09T06:10:00Z",
 "ServerTimestamp": "2026-06-09T06:10:00.412Z",
 "StatusCode": "Good"}
```

Two timestamps, not one: the source's and the server's, which are the Time section's sensor read and gateway record carried by the protocol itself rather than by a convention. A status code that maps onto the tuple's quality. And no unit in the message, because the engineering unit is a property of the node in the address space, read once when the binding is discovered. For the twin that address space is the difference from MODBUS: a reading arrives with its name, its unit and its timestamp attached, so event time is preserved and the binding can be discovered rather than typed.

Kafka is a durable, replayable log behind the gateway, and not a device protocol. In the publish/subscribe evaluation that compares it with brokers, the broker does the heavy lifting; Kafka's subscribers read partitions in order, each message carrying an offset that marks its position, so a subscriber that disconnects and reconnects continues from the last message it read, and many consumer groups can read the same topic at once [17]. The same reading as a record in the log:

```
topic:      fleet.readings     partition: 7     offset: 88213417
key:        wt07/gen-bearing/temp
timestamp:  2026-06-09T06:10:00.655Z        (the broker's, when appended)
value:      {"binding": "wt07/gen-bearing/temp",
```

```
"event_time": "2026-06-09T06:10:00Z",
"value": 61.2, "unit": "degC", "quality": "good"}
```

The payload is the stream's own line, unchanged; what the log adds is outside it. The offset is the log's position, not the publisher's count, so a consumer that resumes from offset 88213416 receives this record again and can tell that it has. That is resumability, ordering and deduplication as properties of the log rather than as conventions in the payload, which is why a durable log sits so often between the adapter and everything else: the data-analytics reference architecture uses Kafka topics as its ingest stream, camera images published to one topic and classification results to another [2]. It is the Processing section's durable buffer.

One handler per protocol, one form upstream

Whatever the transport, the shape of the code is the same: one **handler** per protocol, one normalised form upstream. The handler is where register numbers, multiplexer channels and scaling factors die, and it is the only place a unit is attached. Everything above it sees the tuple. An edge-twin framework shows the shape at scale: devices speaking MQTT, HTTP and CoAP with their own JSON layouts are each shadowed by an instance that homogenises the protocol to MQTT and the data format to one standard, and the overhead of doing so was measured at ten messages a second with hundred-byte payloads and found comparable to the framework's other configurations [18]. Two rules follow. A scale factor is applied in exactly one handler, never in a handler and again in a function, which is the failure the Encodings section works. And the handler is tested with a captured frame and an expected tuple, which is the only test that catches a byte-order error before the turbine's bearing reads minus forty degrees.

The adapter's output contract

What every handler produces, whatever protocol it spoke, is one thing: the **tuple** of six fields the Twinning Loop section named, now fixed as the adapter's output contract.

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

The **binding** names which sensor or channel produced the reading – this turbine, this bearing, this register – and is part of the twin's configuration. The **quantity** says what was measured. The **value** is the number. The **unit** is attached exactly once, here. The **event time** says when the physical event happened, by the best clock available where it happened. The **quality** says how far the value can be trusted, in one of six states the Quality section defines. Chapter 4's shadowing pipeline began with a step called *normalise*; this tuple is what normalise produces, and everything above the adapter – the functions, the shadowing function, the state, every service – reads tuples and nothing else. That is what lets the rest of the twin be written once for every protocol. Look back at the five samples in the Protocols subsection: the MODBUS reply carried two bytes, the OPC UA data value carried two timestamps and a status code, the Kafka record carried an offset, and the handler for each turns what it was given into the same six fields. A bearing temperature that arrived as a MODBUS register and one that arrived as an MQTT message are the same tuple by the time anything else sees them, and the code that reads them does not know which was which.

The adapter's whole output is a stream of tuples, and every section of this chapter is about one of the six fields or about the stream they form: the Sampling and Sensors sections were about the value, the Time section about the event time, this section and the Encodings section about how the stream travels, the Functions and Processing sections about what is computed from it, the Storage section about where it lands, and the Quality section about the sixth field.

Encodings: A Value in Flight

A tuple's value spends its journey encoded, and each encoding is a set of conventions about how a number is laid out in bytes or characters. This section takes one reading from the Transports section's stream – the bearing at 61.2 degrees at 06:10:00 – and encodes it six ways, from the register up, then organises the encodings by one distinction: whether a reader with nothing but the message can tell what is in it.

A register is a convention, applied once

At the device layer a value is a register: a sixteen-bit integer with a scale factor, an offset and a byte order, and every one of those is a convention that must be applied exactly once. The bearing temperature of 61.2 degrees stored as tenths of a degree is the integer 612; stored as hundredths with an offset of minus forty degrees, it is 10,120; stored big-endian it arrives as the bytes `02 64`, little-endian as `64 02`. Applied twice, the scale turns 61.2 into 6.12; applied zero times it stays 612 and looks like a temperature, a hot one; applied with the wrong byte order it becomes 25,602 and looks like a fault. None of these failures is visible in the tuple, because the tuple carries a number and a unit and both look fine. The binding records all three conventions, the handler applies them, and no other code touches them. Two bytes on the wire is the first row of Table 5, and the cheapest encoding there is, because everything but the value is somewhere else.

Text and binary

At the system layer the value is inside a message, and the message is in one of two families. **Text encodings** are readable by a person and verbose on the wire. The reading as JSON is 124 bytes when written on one line:

```
{"binding": "wt07/gen-bearing/temp", "event_time":
 ↪  "2026-06-09T06:10:00Z",
 "value": 61.2, "unit": "degC", "quality": "good"}
```

and as XML, with the fields as attributes, 116:

```
<reading binding="wt07/gen-bearing/temp"
 ↪  event_time="2026-06-09T06:10:00Z"
         unit="degC" quality="good">61.2</reading>
```

Neither needs anything but itself to be read: the field names are in the message, the types are visible from the syntax, and a person can read either in a browser. **Binary encodings** are compact, and how compact depends on what they leave out. The Concise Binary Object Representation (CBOR) is JSON's data model in bytes – the same reading is 101 bytes, and here is how they are spent:

```
A5                                      a map of five pairs
  67 62 69 6E 64 69 6E 67               "binding"                  (7 bytes)
  75 77 74 30 37 2F 67 ... 6D 70        "wt07/gen-bearing/temp"    (21 bytes)
  6A 65 76 65 6E 74 5F 74 69 6D 65      "event_time"               (10 bytes)
  74 32 30 32 36 2D 30 ... 30 5A        "2026-06-09T06:10:00Z"     (20 bytes)
  65 76 61 6C 75 65                     "value"
  FB 40 4E 99 99 99 99 99 9A            61.2 as a 64-bit float
  64 75 6E 69 74   64 64 65 67 43       "unit" "degC"
  67 71 75 61 6C 69 74 79               "quality"
  64 67 6F 6F 64                        "good"
```

CBOR saved 23 bytes on JSON, and the arithmetic says why so few: the five field names are still in the message, the timestamp is still twenty characters of text, and the one number became nine bytes where JSON spent four characters. A binary encoding that keeps the names is still self-describing, and still nearly as large. The saving comes from dropping the names, and dropping the names means both ends must agree on a schema. **Protocol Buffers** does that with numbered fields. The schema:

```
message Reading {
  string  binding    = 1;
  int64   event_time = 2;   // seconds since 1970-01-01T00:00:00Z
  float   value      = 3;
  string  unit       = 4;
  Quality quality    = 5;   // an enum: good = 1, uncertain = 2, ...
}
```

and the reading, 42 bytes:

```
0A 15 77 74 30 37 2F ... 6D 70   field 1, 21 bytes: the binding
10 B8 DC 9E D1 06                field 2: 1780985400, a varint
1D CD CC 74 42                   field 3: 61.2 as a 32-bit float
22 04 64 65 67 43                field 4: "degC"
28 01                            field 5: good
```

Each field is a one-byte tag and a value; the names are in the `.proto` file, not on the wire, and the timestamp is five bytes of integer instead of twenty of text. Replace the binding string with an integer id that the twin's configuration maps back to a name, let the unit be implied by the binding, and the same reading is 16 bytes. **Avro** goes one step further and drops the tags too. Its schema is itself written in JSON:

```
{"type": "record", "name": "Reading", "fields": [
  {"name": "binding",    "type": "string"},
  {"name": "event_time", "type": "long"},
  {"name": "value",      "type": "float"},
  {"name": "unit",       "type": "string"},
  {"name": "quality",    "type": {"type": "enum", "name": "Quality",
     "symbols": ["good", "uncertain", "substituted",
                 "out_of_range", "no_reading", "not_attempted"]}}]}
```

and the reading is 37 bytes: the binding's length and characters, the timestamp as a variable-length integer, four bytes of float, the unit's length and characters, and one byte for `good`, in the schema's order and nothing between them. A reader without that exact schema cannot

even find where one field ends and the next begins.

One number in those examples deserves a second look. Protocol Buffers and Avro carried 61.2 as a 32-bit float, which holds 61.2000008 – a rounding of its own, of about a ten-millionth of a degree. By the Sampling section's rule that is free: the sensor's resolution is a tenth of a degree, and a rounding a million times finer than the sensor's loses nothing the sensor had. A 64-bit float, as CBOR used, buys four more bytes of precision the reading never had.

Table 5 puts the six side by side.

Table 5: One reading of the bearing temperature in six encodings. The bytes fall as what the reader must already know grows; the two-byte register at the top is the cheapest encoding and the one that carries least.

Encoding	Bytes for one reading	What the bytes carry	What the reader must already know
Raw register	2	one integer	the binding: scale, offset, byte order, unit, when it was read
JSON	124	names, values, types by syntax	nothing
XML	116	names and values, as attributes	nothing; a schema may be referenced
CBOR	101	names and typed values	nothing
Protocol Buffers	42, or 16 with an integer binding id	numbered fields and values	the `.proto` file
Avro	37	values in schema order	the writer's exact schema

Read Table 5 down its last column and the saving in the second is bought with the fourth: every byte taken off the wire is a fact moved into a schema somebody must keep. The platform survey lists the family a twin will meet – JSON, CBOR, CSV, XML, Protocol Buffers and YAML among the structured formats – and states why the choice matters: a protocol carries data in an encoding largely dictated by the protocol, the encoding affects the size of the payload and hence the communication latency, and compact formats such as CBOR and Protocol Buffers reduce that overhead [10]. On a constrained link – a radio, a metered cellular connection, a satellite backhaul from a turbine at sea – payload size is latency, and the choice of encoding is a choice about how far behind the twin runs. Three readings an hour at 124 bytes is nothing on any link; two thousand tower accelerations a second at 124 bytes each is two megabits a second before a single reading has been stored, and at 16 bytes it is a quarter of that.

One protocol offers both families of the same message. OPC UA's publish/subscribe part complements its client/server model with a binary and a JSON encoding, and chooses its transport by direction: for lateral connectivity on a local network it uses the User Datagram Protocol (UDP), where a missing packet causes no wait for a correction, which suits time-sensitive networks and constrained devices; for vertical connectivity to cloud environments

it uses MQTT or the Advanced Message Queuing Protocol (AMQP) [19]. One information model, two encodings, two transports, chosen by where the message is going – which is the choice this section is about, made inside one standard.

Self-describing or schema-required

The distinction that organises every encoding: an encoding is either **self-describing**, so that a reader with no other information can recover the field names and types from the message, or **schema-required**, so that the schema must live somewhere the reader can reach. Table 6 sorts the encodings worked above.

Table 6: Encodings sorted by whether the message describes itself. The second is smaller on the wire and the first is smaller in the organisation: a self-describing message costs bytes, a schema-required one costs a place to keep the schema and a rule for changing it.

Encoding	Family	Self-describing?	Where the schema lives
A raw register	binary	no	in the binding: scale, offset, byte order
JSON	text	yes, at the level of names and types	nowhere required; a schema may be published beside the API
XML	text	yes, and the schema may travel with it	in a schema document the message may reference
Protocol Buffers	binary	no	in a `.proto` file both ends compile from
CBOR	binary	yes, at the level of types	as for JSON
Avro	binary	no, per message	in a JSON schema, sent once or kept in a registry

The second family is smaller on the wire and the first is smaller in the organisation, and the choice is about which of the two is scarce. A turbine at sea has scarce bandwidth and a fixed set of quantities: a schema-required encoding, with the schema versioned in the twin's configuration, is the right answer. A greenhouse on a local network with sensors that change every month has scarce engineering time: a self-describing JSON message that a person can read in a browser is the right answer, and the extra bytes cost nothing. When the schema is not in the message it lives in one of three places – compiled into both ends, sent once when a connection opens, or kept in a registry both ends can reach – and each of those is a thing to version, because a schema that changes under a stored message makes the message unreadable. The Avro reading above is the sharpest case: 37 bytes that mean a bearing temperature under one schema and nothing at all under another.

Units as fields or as convention

Units travel either as a field in the message or as a convention between the people who wrote the two ends, and only the first can be checked by a program. A JSON reading with `"unit": "degC"` beside its value can be validated; one whose unit is understood because the sender's documentation says so cannot, and the documentation is the first thing to go stale. A unit that travels as a field should be a code from a list rather than a free string: the AutomationML practical guide recommends the United Nations Economic Commission for Europe (UNECE) Recommendation 20 codes for units of measure, using each unit's common code rather than the unit's own symbol, so that a program can compare two units by comparing two codes [20]. The handler attaches the unit once; a code list is what lets the validate function check it.

Functions: The Computations Between Reading and State

Chapter 4 defined functions as stateless, testable, reusable across twins, and said they were the software engineer's home turf. This section first sorts the functions a twin has by what they do, because only some of them belong to this chapter, and then builds the ones the upward leg needs.

Shadowing, augmentation, and the other things a function does

A twin's functions are named by what they do for the twin, and the survey the platform literature most often builds on lists what a twin does beyond holding a model: it *reflects* its physical object, representing its properties, conditions and behaviours through models and data and remaining its virtual counterpart across the object's whole lifecycle [21]. That reflection needs a link, which the same survey calls *entanglement*: the exchange of status information between the physical object and its logical counterpart, in real or near-real time, with connectivity as its first property and a direction that is typically physical to logical but valuable in both [21]. Around those two it names the rest: *memorisation*, so that the twin can be represented and controlled over a large part or the whole of the life cycle; *composability*, the grouping of several objects into a composed one whose behaviour is observed and controlled as a whole and by component; *augmentation*, which turns an object whose functionalities and services were fixed for its whole life into a programmable one; and *predictability*, which with memorisation lets the twin project its behaviour into the future [21].

The platform work that followed turned two of those into named functions. The **shadowing function** is the one Chapter 4 put in its pipeline: it subscribes to the events arriving from the physical side, processes them against one or more models, and computes and maintains the twin's state [22]; in the Web of Digital Twins the same process also runs downward, propagating an action requested on the twin to the physical asset by a further shadowing function, so that a twin may mirror a lamp's switch as well as its brightness [23]. **Augmentation** is the other: a twin is a software replica whose responsibility is to clone the physical asset's available resources and functionalities *and* to extend existing behaviours with new capabilities, and a design that treats twins as passive data structures, each without its own shadowing process, forgoes exactly the augmentation and composition capabilities the recent literature describes [18]. Table 7 sorts the functions a twin runs by those names, and says which of them this chapter builds.

Table 7: The functions a twin runs, sorted by what each does. This chapter builds the first and fourth rows and reads the second from a distance; the rest are named so that the reader can tell, when a later chapter builds one, which kind it is.

Function type	What it computes	Reads, and writes	Where in this book
Ingestion	valid, clean, transformed tuples from protocol frames	frames and the binding; writes the store	this chapter, below
Shadowing	the twin's belief about the asset, from tuples and a model	the store, the model, the last state; writes the state and `as_of`	Chapter 4's pipeline; the state and estimator chapter
Augmentation	what the asset cannot compute about itself: a residual, a forecast, an alarm	the state and the models; writes derived data and the services	the derive function here; the models and services chapters
Memorisation	the history, kept and summarised	the tuple stream; writes the stores	this chapter's Storage section
Composition	a fleet's belief from its members'	other twins' states; writes a composed state	the manager and platforms chapters
Reflection downward	a command for the asset from a request on the twin	the state and the command path's checks; writes the physical interface	the manager chapter

Read Table 7 for what each row writes and the line this chapter draws is visible. Everything the first and fourth rows write is data: a five-year history of bearing temperatures is data, however it is stored. The twin's current belief about the bearing – its estimated temperature, the time that estimate describes, and whether the twin is currently shadowing the turbine at all – is state, and the second row is the only one that writes it. Chapter 4 said data is not state; this table says which function makes the difference. The ingestion functions built below are the part of shadowing that runs before any model is consulted, and nothing in this chapter writes the state document. It fills the store the shadowing function reads, and the state and estimator chapter is about how well that function reads it.

Five functions, five ways to fail

The ingestion pipeline is five functions, and they fail five different ways. Figure 5 draws them in order.

Ingest turns bytes into a tuple: the handler of the Transports section. It fails by silence – a source that stops, a subscription that lapses – and its test is the heartbeat and the age of twin.

Validate checks the tuple against what the binding says it should be: the value is a number,

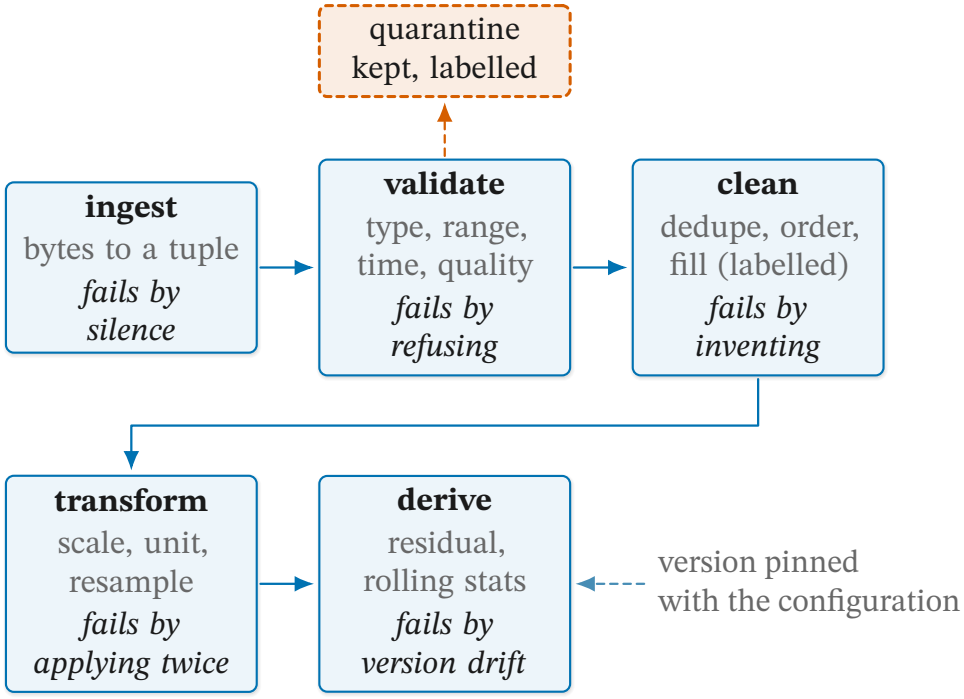


Figure 5: The ingestion pipeline as five functions. Each has a different failure: ingest fails by silence, validate by refusing, clean by inventing, transform by being applied twice, derive by drifting from the version that made yesterday's numbers. A refused tuple goes to quarantine, not to the bin.

inside the sensor's range; the event time is plausible, not 1970 and not tomorrow; the unit matches the binding's; the quality is one of the six states. It fails by refusing, and refusal is its job. Here it is, in the shape it takes for a temperature binding:

```
def validate(t, binding, now):
    if not isinstance(t.value, float):
        return refuse(t, "not a number")
    if not binding.range_lo <= t.value <= binding.range_hi:
        return refuse(t, "out of range")
    if t.event_time > now + binding.clock_slack:
        return refuse(t, "event time in the future")
    if t.event_time < binding.commissioned:
        return refuse(t, "event time before the sensor existed")
    if t.unit != binding.unit:
        return refuse(t, "unit mismatch")
    return accept(t)
```

Every threshold in it comes from the binding, none is a literal, and `refuse` writes the tuple to quarantine with the reason rather than dropping it.

Clean repairs what validate let through but the store cannot take: it removes duplicates, puts out-of-order tuples in event-time order, and fills a short gap if the binding says a gap may be filled. It fails by inventing – a filled value stored as if measured – and the Quality section's `substituted` state is the guard.

Transform applies a scale, converts a unit, resamples a stream to a common interval. It fails

by being applied twice, which is the Encodings section's register turned into 6.12 degrees, and its guard is that a transform records what it did in the tuple's provenance so a second application can see the first.

Derive computes new quantities from clean ones: the residual of Chapter 2, a rolling mean, a rate of change, a spectral peak. It fails by drifting from the version that produced yesterday's numbers, and its guard is the versioning below.

Quarantine, do not discard. A burst of readings that fail validation is a diagnostic signal about the sensor, the link or the firmware – a gateway that rebooted to 1970 produces a thousand refusals in a row, and those thousand tuples, with their reasons, are the evidence of what happened. A pipeline that drops them has thrown away the evidence of its own fault and will report a gap it cannot explain.

Which run where

Chapter 4's table of blocks on layers put functions in two places, the adapter and the core, and the pipeline splits along that line. Decoding, scaling, unit attachment and validation run **in the adapter**, because they need the binding and nothing else, and because the tuple that leaves the adapter must already be a tuple. Residuals, rolling statistics and everything that needs the model or the state run **in the core**, because they read the state. The Industrial Internet Reference Architecture draws the same line from the edge side: the edge performs protocol adaptation, syntactic translation from binary readings to JSON and semantic translation such as unit conversion, and may run edge computing functions such as a fast Fourier transform on vibration data, while the information domain's functions ingest, check quality, augment with metadata, persist and distribute data as it arrives [24]. The placement is not fixed for all time: fluid digital twins relocate computational functions along the edge-to-cloud continuum as conditions change [22], and serverless platforms package a function as a unit the platform places and scales on its own – function as a service, in the cloud-native survey's terms [25]. The Processing section takes up where; here the rule is what: a function that needs only the binding is the adapter's, a function that needs the state is the core's.

Testing and versioning

Test a function with a reading and a number. A validate function is given a tuple with a value of 612 degrees and a binding whose range ends at 150, and is expected to refuse; a transform is given the bytes `02 64`, a big-endian binding with a scale of a tenth, and is expected to return 61.2. There is no ceremony and no fixture beyond the binding, and that is the reason functions are the block where ordinary unit testing applies without argument.

Version them with the configuration, because a threshold is a decision somebody made. When the range in a binding changes, or the residual's formula changes, the derived data in the store changes meaning from that moment, and a row that does not say which version made it cannot be compared with a row that does. Every derived tuple carries the version of the function that made it; every function's parameters live in the configuration Chapter 4 put in version control; and the digital thread of the Vocabulary section is, for functions, exactly this habit.

The check no adapter function can make

One fault defeats every function in this section. A **stuck sensor** reports the same value, forever. The value is a number, inside the range, with a plausible time, in the right unit – valid, clean and well formed – and it is wrong, because the bearing's temperature is not holding to a hundredth of a degree through a day's operation. No adapter function can catch it, because catching it requires knowing what the reading *should* have done, and that is a question for the model. This chapter states the fault, gives it the `uncertain` quality state when a person notices, and hands the detection to the state and estimator chapter, where that estimator compares what arrived with what was predicted and can say that a value that never moves is the one value it did not expect.

Processing: Streams, Windows and Placement

The functions of the previous section run over the stream of tuples, and three decisions govern how: whether they run as the tuples arrive or over a batch later, over which span of tuples, and on which machine.

Stream or batch

A **stream** computation runs on each tuple, or each small window of them, as they arrive; a **batch** computation runs over a stored span later. The rule for choosing: if the answer changes the physical system's behaviour within that system's own time constant, the computation belongs on the stream; if not, a batch is cheaper and easier to test. The turbine's vibration alarm must fire within a second of the tower exceeding its load, because the turbine can be damaged in seconds: stream. Its daily fatigue summary changes a maintenance plan measured in months: batch, run at night over the day's Parquet files. The residual per check interval sits between: a stream computation on a warm path whose window is one check interval.

Windows

A window is the span of a stream over which one result is computed, and there are three shapes. Figure 6 draws them on one timeline.

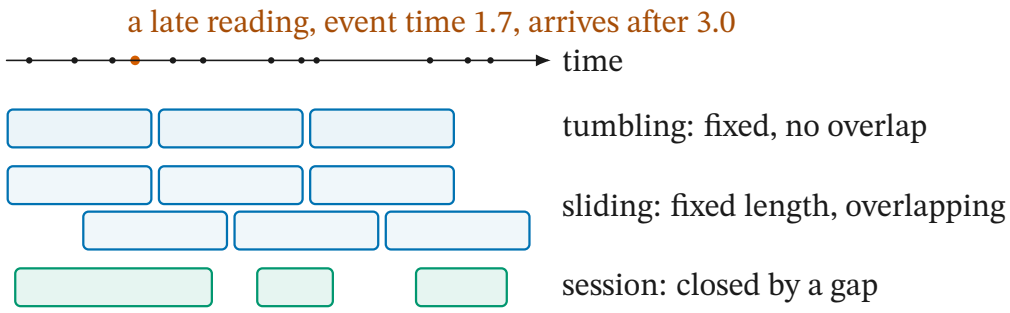


Figure 6: Three windows on one stream. Tumbling windows are fixed and do not overlap, so each reading is in exactly one; sliding windows are fixed and overlap, so each reading is in several; session windows are closed by a gap in the readings rather than by the clock. The late reading belongs to a window that has already closed.

Tumbling windows are fixed in length and do not overlap: the ten-minute mean of the bearing temperature, one result per ten minutes, each reading in exactly one. **Sliding** windows are fixed in length and overlap: a one-hour rolling mean recomputed every minute, each reading in sixty results. **Session** windows are bounded by a gap in activity rather than by the clock: one result per start-up of the turbine, from the first reading after a stop to the last before the next. Choose by the question – a per-interval summary tumbles, a trend slides, a per-episode statistic is a session – and note that the three windows are defined on **event time**, not arrival time, which is what makes the next problem a problem.

Late data

A tuple whose event time falls in a window that has already closed is **late**, and every stream system does one of three things with it. **Drop** it. Hold each window open for a **grace period** after its end and accept late tuples until then. Or **emit corrections**: close the window on time, and when a late tuple arrives, recompute and publish the window's result again with a note that it supersedes the first.

Drop is the silent default. A window that closed before a reading arrived, in a system that never said what it does with late readings, has dropped it, and the record shows nothing. The gateway that buffered an hour through an outage and replayed it delivers an hour of late tuples, and a pipeline built without a late-data policy discards the hour while reporting every window on time. The policy belongs in the configuration, per binding, because the right answer differs: a hot-path alarm drops, because a late alarm is no alarm; a warm-path residual uses a grace period of one check interval; a cold-path summary emits corrections, because a fatigue total that omits an hour is wrong for the rest of the turbine's life.

Edge, fog and cloud

Where a function runs is a placement decision on a continuum: on or beside the asset (the **edge**), on a site or regional node (the **fog**), or in a datacentre (the **cloud**). The placement literature treats it as an optimisation with measured constraints: one study places the components of a componentised application across edge, fog and cloud segments, deploys the result on a testbed, and derives guidelines under infrastructure constraints and response-time targets [26]; the twin middleware work envisions a twin as an ecosystem of microservices and serverless functions whose activation, mobility and replication along the continuum are driven by cyber-physical metrics, and observes that mature orchestration for that is still being sought [27]. There is measured evidence that offloading can win: a model of edge-only against edge-to-cloud completion time, as a function of link speed, task complexity and edge processor load, shows conditions under which shipping a big-data job to the cloud finishes sooner than running it at the edge, and conditions under which it does not [28]. For the upward leg the rule of thumb follows the paths: hot-path functions run at the edge, because the round trip to the cloud is longer than the budget; cold-path functions run in the cloud, because the data is already there and the compute is cheap; and warm-path functions go wherever the bandwidth argument of the Storage section allows.

The durable buffer

Between the adapter and everything downstream sits, in most twins of any size, a durable log – the Kafka of the Transports section or something like it. Its job on the upward leg is

to make a slow consumer harmless: because the log keeps every tuple and each consumer reads from its own offset, a consumer that falls behind or restarts continues from where it was and loses nothing [17], and because the adapter writes to the log rather than to the consumer, a slow consumer never applies back-pressure to a sensor. Without the buffer a slow dashboard slows the adapter, and a slowed adapter drops readings, and the record has a gap whose cause was a chart. With it, the gap becomes a lag, measured by the age of twin, and lags are recoverable where gaps are not.

State estimation – combining the stream with a model to produce the state Chapter 4 put at the centre – is the next computation the stream feeds, and it is the state and estimator chapter's subject. This section ends where the tuples are clean, windowed, placed and buffered; that chapter begins by reading them.

Storing Twin Data

One twin, several stores, and none of them owned by the twin: Chapter 4 said the store is a neighbour the twin reads and writes, and this section says how to lay it out. It carries the Transports section's bearing temperatures through every store they land in, so that the same three readings can be seen as rows, as columns, as a point in a time-series database, as an object in a bucket, and as the metadata record that says what they mean.

Several stores, none owned

A twin of the turbine has, at the least, a time-series database for the hot and warm paths, an object store for the cold path and for anything that is not a row, and a metadata store for what the readings mean. Each is read by other software than the twin and outlives any one version of it. The volume is in the first two, the meaning is in the third, and the shape the data takes in each – rows, columns, arrays – is the first thing to settle.

Rows and columns

Most telemetry is a table: one row per reading, columns for the tuple's fields. The three readings as a table are three rows under one header:

```
binding,event_time,value,unit,quality
wt07/gen-bearing/temp,2026-06-09T06:00:00Z,61.1,degC,good
wt07/gen-bearing/temp,2026-06-09T06:10:00Z,61.2,degC,good
wt07/gen-bearing/temp,2026-06-09T06:20:00Z,61.4,degC,good
```

That is a comma-separated values file (**CSV**), the universal export and the interface a supervisory control and data acquisition (SCADA) system will give you when nothing else is offered. It has no types, no units and no schema beyond its header row – nothing in the file says that `61.1` is a number and `degC` is not – and every one of those is supplied by the person who reads it. Treat a CSV export as a transport in disguise – resumable if the file is dated, ordered only if sorted – and land it in something better. JSON Lines, one JSON object per line, is the same idea with types and names on every row; the Transports section's stream was written in it, and it is what many pipelines write when they need an append-only text file.

The same three rows, stored by column instead:

```
event_time:  06:00:00Z   06:10:00Z   06:20:00Z   ...   144 values a day
value:       61.1        61.2        61.4        ...   neighbours alike
quality:     good        good        good        ...   one distinct value
binding:     wt07/gen-bearing/temp                     stored once
```

A row-oriented store keeps each reading's fields together, so writing one reading is one write and reading one reading is one read. A column-oriented store keeps each field's values together, so the mean of one field over a month touches one column and skips the rest. Telemetry is columnar at rest because the questions asked of it are columnar: nobody asks for "the reading at 06:10:00", everybody asks for "the bearing temperature over June", and a column of temperatures compresses far better than a row of mixed fields because its neighbours resemble each other – the value column above changes by a tenth of a degree per entry, the quality column does not change at all, and the binding is written once for the whole file. The hot path, which writes one reading at a time and reads the latest, is row-shaped; the cold path, which writes once and reads columns, is column-shaped; and a twin holds both.

Parquet is the columnar file the cold path lands in. Turbine owners generally store their sensor data as Parquet files in an object store and analyse it with Julia, Matlab, Python or R [29]; the ModelarDB system writes Parquet or Optimized Row Columnar (ORC) files to the local file system at the edge and to a distributed file system in the cloud [30]; and practitioners' first answer to compressing high-frequency turbine data is lossless big-data formats such as Parquet or ORC written at the edge and transferred [31]. **Arrow** is the in-memory column format that lets tools share data without copying. The same time-series system returns query results in Arrow so that programs can consume them directly, and transfers compressed segments between its edge and cloud instances over Arrow's own transport protocol rather than as rows [30].

Arrays and fields

Not everything is a row. Some data is a grid, an image or a cloud of points, and forcing it into rows destroys the structure a model needs.

Gridded environmental data – temperature over a sea surface, rainfall over a catchment – is a multidimensional array with named axes, and NetCDF is its file format: a marine twin of a coastal lagoon gathers its feeds from several providers, one of which delivers its datasets in NetCDF over HTTP [32]. **Labelled multidimensional arrays** are the working form for simulation inputs and outputs: a plant-modelling environment builds its inputs and outputs as labelled arrays and data sets so that index- and label-based selection and interpolation are available on every one of them, and stores its parameter values the same way [33]. **Raw video** is an integer array with sidecar metadata: high-speed camera recordings in structural dynamics reach hundreds of gigabytes and are decoded from proprietary formats into a three-dimensional array of images, height and width, read as a memory-mapped unsigned sixteen-bit array beside a dictionary of metadata from a sidecar file [34]. **Point clouds** from 3D scanners are product-quality data: the manufacturing review names coordinate measuring machines, 3D scanners and computer tomography as the technologies that measure how well parts meet their specification, with one study storing point clouds of individual components and their deviation from nominal [9]. **Images** travel as files: the data-analytics reference architecture publishes camera images as compressed image files on a Kafka topic and consumes them from it [2].

What these share is that the value is not a number but an array, and the tuple's `value` field holds a reference – a file path, an object key – with the array's shape, type and axes in the metadata beside it. The event time, binding and quality fields are unchanged. A thermal image of the bearing housing at 06:10:00 is still a tuple; its value is an object in the store the Object storage subsection describes.

The time-series database

Chapter 2 introduced the shape when it described InfluxDB's point: a **measurement** name, **tags**, **fields** and a **timestamp**. The shape is general to time-series databases, and the reading at 06:10:00 takes it like this:

```
measurement:  bearing_temp
tags:         turbine=wt07   component=gen-bearing   quality=good
fields:       value=61.2
timestamp:    2026-06-09T06:10:00Z
```

The one design decision is which of the tuple's parts became tags and which became fields. Tags are what you filter by, and the store indexes them; fields are what you aggregate, and the store scans them. The binding's parts – turbine, component – are tags, because the questions are "this turbine's bearings" and "all turbines' generator-side bearings". The value is a field. The quality is a tag if you filter on it ("only good readings") and a field if you do not; here it is a tag. The unit is in neither: it is fixed per measurement and lives in the metadata below. A fleet of two hundred turbines with a handful of monitored components each makes the `turbine` and `component` tags a few hundred distinct pairs, and the index stays that size for the life of the fleet.

The high-cardinality tag is the mistake that is hard to reverse. A tag with a new value on every message – a message identifier, the MQTT sample's `seq`, the event time itself – gives the store a new distinct value to index on every write, so the index grows with the data instead of with the fleet: 144 new series per binding per day for this stream, and the store slows and swells until the tag is removed, which means rewriting the data. The test before making anything a tag: count its distinct values, and ask whether the count is the size of the fleet or the size of the history.

Retention and downsampling

A store keeps raw points for a **retention** period and then either drops them or replaces them with summaries – **downsampling**. Chapter 2's rig keeps seven days and drops; a turbine's twin keeps the raw stream for weeks at the edge and summaries for years in the cloud. Turbine owners' second answer to volume, after lossless files, is exactly this: keep simple aggregates such as ten-minute means at the edge, which is fast and compresses well and loses quality with no bound on the loss [31].

The loss that matters is which summary is kept. Downsampling that keeps only the mean destroys the very thing a fault detector needs: a vibration spike that lasted a second is averaged into a ten-minute mean and vanishes, and the record of the month shows a calm turbine. Keep min, max and count beside the mean, so that a spike survives the summary that averaged it away and a gap shows as a count below the expected number of readings. After thirty days the three readings above and the three that followed them become one row

for the hour:

```
bearing_temp  turbine=wt07 component=gen-bearing
  mean=61.3  min=61.1  max=61.6  count=6      2026-06-09T06:00:00Z
```

Four numbers per window instead of one, and the month's record can still say that something happened on the ninth: a `max` far above the `mean` is a spike, a `count` of five is a missing reading.

Model-based compression within a bound

There is a third answer to volume, between lossless files and lossy means. **Model-based compression** represents a span of readings by a fitted model – a constant, a line, a small polynomial – within a declared error bound, and stores the model's parameters instead of the points. The three readings climb by a tenth of a degree per ten minutes and a fourth at 61.5 would too; a line through them, stored as a start value, a slope and a span, reproduces all four within the sensor's resolution and replaces four points by three numbers. The time-series system built for turbine data does this at scale: it compresses groups of series within a user-defined pointwise error bound, which may be zero, by trying several model types and keeping the one that compresses best, falling back to a lossless encoding where the values move more than the bound allows [30]. On real turbine data it outperformed the lossless big-data formats even at a zero bound, a tiny bound of one hundredth of a per cent improved compression significantly, and at a five per cent bound it was nearly thirty times smaller than the lossless files [31].

Why a bounded loss can be free: recall Equation 1. The ten-bit temperature reading was already rounded to a third of a degree by its converter, so a compression that reconstructs it within a tenth of a degree has lost nothing the sensor ever had. A bounded loss smaller than the sensor's own quantisation step is free, and the datasheet's resolution row is where the bound comes from. The same system's authors note that the edge hardware at renewable installations resembles a low-end commodity desktop computer, so a high compression ratio is worth more there than a low processor load [30] – the Sampling section's ceiling, set by the gateway, showing up in the store.

Object storage: the cold path

The cold path in one paragraph: Parquet files in an object store, partitioned by time and then by asset, with the raw data kept so that it can be reprocessed when a function's version changes. An object store holds named blobs and answers two questions quickly – give me this key, list the keys under this prefix – and nothing else, so the layout of the keys is the whole of its design. The day's readings for the bearing land under

```
telemetry/year=2026/month=06/day=09/turbine=wt07/bearing_temp.parquet
```

with the 144 readings of the day as columns inside it. Partition by time first because every cold-path query is bounded by a span – June, last year – and a time-first layout lets the query skip whole prefixes; by asset second because the next filter is always "this turbine". Keep the raw data because the derive functions will change, and a summary computed by version 3 of the residual cannot be recomputed by version 4 from the summary. Industrial cyber-physical systems have adopted data lakes on distributed file systems, and what those lakes hold is raw data [35]; the turbine systems above write their compressed segments

continuously to an object store in the cloud [29]. What an object store cannot answer quickly is "the latest reading", because finding it means listing keys and opening a file; that is the time-series database's question, which is the reason a twin holds both.

Metadata

Every store above holds the readings; none of them holds what the readings mean. The 612 in the register, the 61.2 in the column and the `value` field of the point are the same number, and nothing about any of them says that it is a temperature in degrees Celsius of the generator-side bearing of turbine 7, read every ten minutes by a gateway on network time from a sensor accurate to half a degree and last calibrated in March. That is **metadata**, and for the bearing stream it is one record:

```
{"binding": "wt07/gen-bearing/temp",
 "device": "wt07-plc-1", "register": 42,
 "scale": 0.1, "offset": 0.0, "byte_order": "big-endian",
 "quantity": "bearing temperature", "unit": "degC",
 "range": [-40.0, 150.0], "resolution": 0.1, "accuracy": 0.5,
 "sampling_interval_s": 600, "clock_source": "gateway, network time",
 "calibrated": "2026-03-14", "firmware": "4.2.1",
 "schema_version": 3,
 "valid_from": "2026-03-14T00:00:00Z"}
```

Read it against the chapter: the first three lines are the binding the Transports section's handler applies; the next two are the Sensors section's datasheet numbers, which the validate function reads its range from; the next is the Sampling and Time sections' interval and clock; `schema_version` is what the Encodings section said must be kept when the schema is not in the message; and `firmware` is the Vocabulary section's digital thread, written down. The last line is the one that makes it metadata rather than configuration: a reading is interpreted by the record that was valid at its event time, so when the sensor is recalibrated or the firmware changes, a new record with a new `valid_from` is written and the old one is kept. A reading from March is then still readable in November after both have changed. The record is small, read rarely and versioned in the configuration Chapter 4 put under version control; it lives in a document store or a table keyed by binding, and the arrays of the Arrays and fields subsection carry the same thing as a sidecar beside each object. Without it, every store in this section is a store of numbers.

Sizing the volume

Size the volume before it is a problem, from the sampling interval, the number of bindings and the bytes per point. A published figure gives the arithmetic: a wind turbine generating 2,500 time series sampled at 100 Hz produces over 321 GiB per day if timestamps and values take eight bytes each [31]. Check it:

$$V_{\text{day}} = 2500 \times 100 \times 86{,}400 \times 16 = 3.456 \times 10^{11} \tag{4}$$

bytes, which is 321.9 GiB. Equation 4 is one turbine, one day, raw. Now set it against the link. The same paper reports that bandwidth between edge and cloud can be as low as 512 kbit/s [31]; a day's raw data is $3.456 \times 10^{11} \times 8/86{,}400 = 3.2 \times 10^{7}$ bits per second averaged

over the day, thirty-two megabits a second against a link of half a megabit – more than sixty times too much. That single division decides the architecture: the raw high-rate streams cannot leave the turbine, so the edge stores them, compresses them within a bound, and sends segments; the twin in the cloud sees compressed segments and summaries, and the faded turbine example works the consequence. Do this division for every twin on the day the sampling intervals are chosen, because it is the cheapest architectural decision in the chapter and the most expensive to discover late.

Quality: The Field the Pipeline Carries

The tuple's sixth field has waited until the pipeline that fills it was built. **Quality** is the adapter's statement about how far the value may be trusted, and it takes one of six states.

Six states

Table 8 names the six and what each does downstream, in the three components that read it: the estimator of the state and estimator chapter, the calibration that fits the model to the data, and the alerting service.

Table 8: The six quality states and what each does in the three components that read it. The fourth and fifth rows say nothing about the asset and everything about the instrument; the last two are different failures pointing at different teams.

State	Meaning	Estimator	Calibration	Alerting
`good`	measured, in range, on time	uses it	fits to it	alerts on it
`uncertain`	measured, but the sensor is suspect: drift, disagreement, stuck	uses it with a wider variance	excludes it	alerts, with the state shown
`substituted`	filled in, not measured: an interpolation, a last-known value	uses it as a prior, not an observation	must exclude it	never alerts on it alone
`out of range`	measured, outside the sensor's stated range	ignores the value, notes the event	excludes it	alerts on the sensor, not the asset
`no reading`	the adapter asked and nothing came back	none; the age of twin grows	none	alerts on the link or sensor
`not attempted`	the adapter did not ask: not bound, paused, misconfigured	none	none	alerts on the twin itself

Two rows need a sentence. A substituted value must never lose its label. The moment a filled-in number is stored as if it were measured, the calibration that later fits to it is fitting to the twin's own guess, and the model learns to predict its own interpolation. The clean function may fill a gap; it may never fill it with a `good`. And `no reading` and `not attempted` are different failures pointing at different teams: the first says the sensor or the link failed and belongs to whoever owns the hardware; the second says the adapter never asked – the binding is unbound, the poll loop is paused, the configuration names a sensor that was removed – and belongs to whoever owns the twin. A monitoring service that merges the two into "missing" has thrown away the one fact that says who to call.

What the adapter can decide

The adapter can decide quality from two sources and no more: the sensor and the transport. From the sensor it can read a status register or a confidence the sensor itself reports; from the transport it can see a timeout, a checksum failure, a sequence gap, a will message. Those give it `out of range`, `no reading`, `not attempted`, a `substituted` label on anything the clean function filled, and `uncertain` where a second sensor disagrees or a calibration date has passed. What it cannot decide is whether a valid, in-range, on-time reading is *plausible*, because plausibility needs a model of what the reading should have been – the stuck sensor of the Functions section, the temperature that rose faster than any bearing can. That quality is decided in the core, by the state and estimator chapter, and the adapter's honest state for a reading it cannot judge is `good`, meaning "nothing I can see is wrong".

Gaps, duplicates, out of order

Three defects in the stream itself are the adapter's and the clean function's to detect, and Table 9 gives each its detection and its cost.

Table 9: Gaps, duplicates and out-of-order arrival: how each is detected in the adapter or the clean function, and what each costs when it is not. All three are properties of the transport that the tuple's event time and a sequence number make visible.

Defect	How it is detected	What it costs if missed
Gap	a sequence number skips; the interval between event times exceeds T_s by more than the slack; a heartbeat is missed	a window computed over fewer readings than it claims; a trend fitted across a hole
Duplicate	two tuples with the same binding and event time; a repeated sequence number after a reconnect	every count and sum doubled for the span; a spike that is only a retry
Out of order	an event time earlier than the last one stored for the binding	a correction applied before the reading it corrects; a rate of change with the wrong sign

The operational-metrics study of the Time section is the experimental form of this table:

packet loss with jitter, replay and out-of-order delivery each left a measurable mark on the twin's timeliness, reliability and consistency metrics [8]. The sensor-side counterpart is the drift and calibration trouble of the Sensors section, which reaches the store as noise and inconsistency [6] and is the reason `uncertain` exists.

How to Choose

Everything above collapses into one table and one checklist. The table is organised by the path of the Vocabulary section, because the path is the decision that fixes the rest.

Table 10: How to choose, by path. Assign a service to a path and read down its column; every row is a decision the path has already made.

	Hot path	Warm path	Cold path
Latency budget	milliseconds to seconds	seconds to minutes	hours to months
Sensing rate	at the ceiling the event allows	derived from the check interval	whatever the hot and warm paths produced
Transport	a broker at the edge, or a direct subscription	a durable log	files: Parquet segments over a metered link
Encoding	binary, schema-required	binary or JSON, with a sequence number	columnar files
Layout at rest	rows: the latest reading per binding	rows and short windows	columns, partitioned by time then asset
Placement	edge	edge or fog	cloud
Store	in memory, or a time-series store with short retention	a time-series store with downsampling	an object store, raw kept
Late data	drop	grace period	emit corrections

Read Table 10 down a column, not across a row: a hot-path alarm is a binary message on an edge broker into an in-memory latest-value store with late data dropped, and none of those choices needs separate argument once the path is chosen.

Then the checklist, in order of leverage – each question changes more of the design than the one after it:

1. **What is the latency budget?** The decision's latency from Chapter 1, which chooses the path, which chooses the column.
2. **Who initiates?** Whether the source can push, or must be polled, and whether either is resumable.
3. **How many consumers, and do they grow?** One consumer can poll; ten need a broker; ten that will become a hundred need a durable log.

4. **What are the power and bandwidth?** A battery-fed gateway and a metered link push toward binary encodings, long sampling intervals and compression at the edge.
5. **What must the payload mean to other systems?** If another team's software will read the messages, the encoding must be self-describing or the schema must be published, and the units must be a code list.

Two cautions close the section. Published protocol benchmarks use workloads unlike real devices: the benchmark literature's own critique is that existing publish/subscribe benchmarks stress systems with static configurations of homogeneous publishers, where real IoT devices publish at heterogeneous rates, with varied payload sizes, under intermittent connectivity [36]. And latency is a property of the deployment, not of the protocol: a measurement study across the network stack found that core routing, the access network, the transport protocol and the application protocol each contribute, that a new connection costs a handshake of about eighteen milliseconds and a security setup of about twenty-eight more, and that moving computation close to the user is only a small part of the latency problem [37]. The same protocol on the same hardware gives different numbers on a different network. Measure your own.

The Plant Demonstrator's Data Path

Everything above used the turbine, a building or a generic instance. This section applies the chapter, in order, to the one system the previous chapters put on the desk, and it is the only place the plant demonstrator appears in this chapter. It is built against Chapter 2's API and Chapter 3's stages, and it makes no claim about the rig beyond what Chapter 2 established.

The rig's sensors, read as a designer

Recall Chapter 2's table of quantities. The STEMMA soil sensor reports a ten-bit count scaled linearly to a percentage, every $T_{\mathrm{r}} = 10$ seconds in the shipped configuration; the light sensor reports nine wavelength bands every fifteen seconds; the air sensor reports temperature and humidity at a configurable interval, with a stated confidence of plus or minus 0.2 degrees; the RS-485 probe reports moisture, temperature and conductivity from three registers; and the controller records a `watering` event per dose. Read those as a designer with the Sensors section in hand.

The soil sensor's per cent is a relative scale, as Chapter 2 warned: its range is the converter's ten bits, its resolution is one count in 1,024, and its accuracy against water content is unstated, so the datasheet's accuracy row is empty and the `uncertain` state is the honest quality for it until the calibration chapter fills that row. Its binding is not one address but a pair – the multiplexer's channel, then the sensor's address – so the multiplexer channel is part of the binding, and a sensor moved to another channel is a new binding, not a changed value. One poll of the light sensor produces nine tuples, one per band, sharing one event time. And the watering events are ingest, not actuation: the controller records what it did, and the twin reads that record through the same API as a reading, with quality `good` and a unit of millilitres. Chapter 2's warning stands – that column is a record of the pump schedule, not of the pot.

The REST API against the five properties

Score the API of Chapter 2 against Table 4. Event time preserved: yes, every point carries the `time` the controller stored it at, which is the gateway record of the Time section and the best event time the rig offers. Connection state observable: yes, every `GET` returns a status code or fails, and `GET /` is a liveness check. Gap survivable: yes, `since_timestamp` on the sensing and watering endpoints returns everything since a given time. Ordering knowable: yes, points come newest first with their times. Deduplication possible: yes, unit, parameter and time identify a point. Five out of five, and the one parameter that earns the score is `since_timestamp`: it makes polling resumable, so a twin that was down for an hour asks for the hour and gets it – within Chapter 2's seven-day retention, which is the one limit on the property.

Bindings and the tuple mapping

Table 11 maps the rig's quantities to bindings and tuples.

Table 11: The rig's bindings and the tuple each produces. The multiplexer channel is inside the first binding's configuration, not in the tuple; the nine light bands are nine bindings sharing one poll.

Binding, and the endpoint behind it	Quantity and unit	Event time	Quality the adapter can decide
soil moisture of `plant_1`, from `/sensing/plant_1/moisture`	soil moisture, relative, in per cent of full scale	the point's `time`	`uncertain` (uncalibrated scale)
soil conductivity of `plant_1`, from the `soil_electrical_conductivity` parameter	conductivity, in microsiemens per centimetre	the point's `time`	`good`
air temperature of the greenhouse, from the `air_temperature` parameter	air temperature, in degrees Celsius	the point's `time`	`good`, with the stated confidence carried
nine light bands of the greenhouse, from `light_level_415nm` and eight more	photon flux per band, in photons per second	one `time` for nine tuples	`good`
watering of `plant_1`, from `/actuation/plant_1/watering_events`	dose delivered, in millilitres	the event's `time`	`good`; an event, not a sample

The encoding at each hop

The soil reading is a ten-bit integer in an I2C register – a number at an address, schema in the driver. On the Raspberry Pi it becomes a Python float, then an InfluxDB point with the value, its units and a confidence as fields. Through the API it becomes a JSON object, `{"time": ..., "physical_unit": "plant_1", "parameter": "moisture", "value": 41.2, "units": "%", ...}` – self-describing, with the unit as a field, which is

why the adapter can validate it. In the adapter it becomes the tuple. Three encodings, one scale factor, applied once, on the Pi; the adapter applies none, and its transform function for this binding is the identity.

The poll loop and its watermark

The adapter polls each binding at its own interval with `since_timestamp` set to a **watermark**: the event time up to which everything has been durably stored in the twin's own store. The loop is: read since the watermark; validate each point; write the accepted tuples to the store; and only when the write has succeeded, advance the watermark to the newest event time written. A watermark that advances on a successful read loses every reading the write then fails to keep. The loop is resumable by construction – after an outage it asks since the old watermark – and idempotent, because a point returned twice has the same unit, parameter and time and the clean function drops the second.

The numbers

Table 12 sizes the path for one pot, from Chapter 2's intervals and an assumed sixty-second air interval, with a JSON object of about 150 bytes per point.

Table 12: The rig's data path for one pot and one day. Ten megabytes a day is a hundred kilobits a second averaged over the day – nothing, on a desk network – and the light sensor is four fifths of it.

Quantity	Interval	Points per day	Bytes per day, as JSON
soil moisture	10 s	86,400/10 = 8640	about 1.3 MB
light, nine bands	15 s	$5760 \times 9 = 51{,}840$	about 7.8 MB
air temperature and humidity	60 s, assumed	$1440 \times 2 = 2880$	about 0.4 MB
watering events	per dose	two or three	negligible
total for one pot		about 63,000	under 10 MB

Two more numbers. The age of twin, by Equation 3, depends on which reading the state rests on: if the adapter polls the moisture every $T_{\mathrm{r}} = 10$ seconds, A sits between zero and about ten seconds plus the poll's latency, and a fault detector checking every $T_{\mathrm{c}} = 6$ hours has a state that is at most a few seconds old when it reads it. That is far better than the decision needs; the sampling section's five steps would say the moisture could be read every few minutes with no loss to the decision, and the aliasing check has not been run to say so.

Limitations the credibility chapter will inherit

Five, stated plainly. The soil sensor's accuracy against water content is unknown, so every moisture tuple is `uncertain` until calibrated. No aliasing check has been run on any of the rig's quantities; the ten- and fifteen-second intervals were chosen as reasonable, not derived. The event time is the controller's record time, not the sensor's read time, and the gap between them is unmeasured. A stuck soil sensor is undetectable on this path, because nothing here has a model of how fast a pot can dry. And the seven-day retention below the API bounds the gap the twin can survive. Each of these is a sentence the credibility chapter will have to carry into its argument about how far the rig's twin can be believed.

Faded Examples

Three, decreasing in how much is worked, none of them the demonstrator.

Example 1, worked in full: the offshore turbine's ingest

The turbine that has carried this chapter, made concrete. It is consistent with the models chapter's running example, which uses the same turbine.

Three sampling rates and a fourth timescale. The turbine's SCADA system reports operational signals – wind speed, power, rotor and generator speed, and the temperatures of the shaft bearing, brakes, generator and gearbox oil – at a rate that averages a few tenths of a hertz on a research turbine and is often available only as ten-minute statistics on a commercial one [38], [39]. Its condition-monitoring system samples bearing and gearbox vibration at hertz to tens of hertz. Its structural sensors – tower and blade accelerometers – sample at kilohertz. And its fatigue summary is computed once a day. Four timescales, four decisions: a ten-minute mean serves "is the turbine producing as expected", a hertz stream serves "is a bearing degrading over weeks", a kilohertz stream serves "did the tower exceed its design load in this gust", and the daily summary serves "when does this turbine need inspection".

Clocks. Apply the ratio of the Time section. The ten-minute SCADA statistics tolerate a clock a second wrong. The two-kilohertz accelerometers do not: one second is two thousand samples, and comparing two accelerometers to find a vibration's direction needs the two clocks within a sample of each other, which is hundreds of microseconds. Network time cannot hold that; precision time over the turbine's wired network, or a satellite reference at the gateway, can. That is a purchase, and this paragraph is where it is justified.

Volume and the boundary. By Equation 4, a turbine with a few thousand series at a hundred hertz produces over 321 GiB a day, against an edge-to-cloud link that may be as slow as half a megabit a second [31]. The high-rate streams do not leave the turbine intact. So the boundary sits at the edge: the gateway in the turbine ingests everything, runs the hot-path alarms locally, compresses the streams within a bound set from each sensor's resolution, and sends compressed segments and per-window summaries to the cloud as Parquet, where the turbine owners' object store already lives [29]. The twin in the cloud sees the ten-minute statistics in full, the hertz streams within their bound, and the kilohertz streams as summaries and as segments it can request. The raw kilohertz data stays on the turbine for as long as its disk allows, and a request for it is a cold-path job.

Scoring the transports. The SCADA export is a CSV file per day: gap survivable and ordered, connection state unobservable, deduplication by timestamp. The condition-monitoring

system publishes MQTT with a sequence number and a timestamp in every payload and a will message registered at connection: all five, by convention. The accelerometers write to the edge's local store directly: all five, because the adapter is the writer. Three transports, one tuple, and the age of twin is computed for each.

Example 2, one step left to you: a battery bus fleet

A city runs two hundred electric buses. Each bus's battery management system takes voltage, current and temperature from every cell in the pack every one to five seconds and estimates the pack's state of charge and health, neither of which can be measured directly [40]. The buses are mobile and intermittently connected: in the depot they have wireless network access, on the route they have a metered cellular link that drops in tunnels and at the edge of the city.

The data path is mostly settled by those two facts. The bus is the edge, and it stores everything it measures; the transport is store-and-forward, with the bus publishing to a depot broker whatever it could not send on the route; late data is therefore central rather than exceptional, and every window in the fleet twin must have a grace period or emit corrections; and the cell voltages are the smooth, high-volume channel – a few hundred cells at a hertz – for which model-based compression within a bound set by the voltage sensor's resolution is the right answer, while the current, which spikes with every acceleration, is kept at full rate.

Your step: the fleet twin's state-of-health estimate is recomputed nightly from the day's data, and a bus that was out of coverage for the last hour of its route uploads that hour at 01:00, after the nightly computation has run. Say which late-data policy the nightly computation must use, what the store must keep for that policy to be possible, and what the age of twin for that bus should have shown at midnight. A check appears in the exercise solutions.

Example 3, posed as a problem: a district heating network

A district heating utility has three generations of equipment from three vendors. The newest substations publish MQTT with timestamps from gateways whose clocks are known to drift by minutes. The older substations speak MODBUS to a SCADA system that will export a CSV file of ten-minute averages once a day. The oldest have no telemetry at all, and a technician reads their meters by hand once a quarter and types the readings into a spreadsheet. The utility wants one twin of the network.

Design its upward leg, in eight parts: (1) the tuple that all three sources are normalised to, and what the quality field says for each; (2) the event time each source can offer, and which of the four timestamp questions each fails; (3) how the MQTT clock drift is detected and what is done with a reading whose event time is impossible; (4) the transport score of each source against the five properties, and what convention or component repairs each missing one; (5) the sampling interval each source imposes, and which decisions of the utility's each interval can and cannot support; (6) the store and layout for the quarterly manual readings, and whether they are sampled data or events; (7) the store's schema – which parts of the binding are tags, and what retention each source needs; (8) the age of twin per substation, and what the twin's dashboard should show for a substation whose newest reading is three months old.

Summary

Back to the objectives this chapter opened with.

1. **Read the datasheet for six numbers.** Range, resolution, accuracy, drift, response time and maximum sample rate, each changing something downstream; resolution is how finely a sensor reports and accuracy is how far it may be wrong, and a threshold finer than the accuracy fires on the sensor. Calibration is the number the datasheet cannot give and the cost most often skipped.
2. **Derive the interval from the decision.** Five steps – the decision, the event, several samples across it, the ceiling, the floor – give the sampling interval; aliasing is fast behaviour appearing as slow behaviour, checked by sampling fast once and decimating; the theorem fits on one line; quantisation puts a floor of half a count under every use of a reading. Sample the state, capture the events, align them on time.
3. **Ask five things of a transport.** Event time preserved, connection state observable, gap survivable, ordering knowable, deduplication possible – and before all five, can I ask you for what I missed. HTTP with a history parameter, MQTT with a sequence number and a will message by convention, OPC UA with its address space and two timestamps, and Kafka with its offsets each answer differently; MODBUS carries two bytes and needs a gateway to answer for it. One handler per protocol, one tuple upstream, and the tuple of six fields is the adapter's output contract.
4. **Carry two times and compute three numbers.** The tuple carries event time, the adapter records ingest time, and their difference is a measurement; clock error matters as a fraction of the sampling interval; twinning rate, age of twin and time discrepancy per binding say whether the twin is current, how current, and whose fault it is not. Age of twin is the best single health number a twin has, and `as_of` is what makes it computable.
5. **Encode in flight, count the bytes.** A register is three conventions applied once; one reading is 124 bytes as JSON, 101 as CBOR, 42 as Protocol Buffers, 37 as Avro and 2 as a register, and every byte saved is a fact moved into a schema somebody must keep; encodings are self-describing or schema-required, and the choice is about which is scarce, bytes or engineering time; units travel as a code or cannot be checked.
6. **Sort the functions, then build five of them.** Shadowing computes state from readings and a model, augmentation computes what the asset cannot about itself, memorisation keeps the history; this chapter's ingestion functions are the part of shadowing that runs before any model. Ingest, validate, clean, transform, derive fail by silence, refusal, invention, double application and version drift; quarantine rather than discard; the adapter runs what needs only the binding and the core runs what needs the state; test with a reading and a number, version with the configuration. The stuck sensor is the check no adapter function can make.
7. **Lay the readings out five ways and size the volume first.** Rows for the latest, columns for the history; tags are filtered, fields are aggregated, and a high-cardinality tag is hard to undo; keep min, max and count beside the mean; a bounded loss smaller than the sensor's quantisation is free; Parquet in an object store keyed by time then asset, raw kept; a versioned metadata record beside it all, without which every store is a store of numbers; and one division – bytes per day against bits per second on the link – decides where the boundary sits. On the rig, the whole path is ten megabytes a day and `since_timestamp` earns the API five out of five.

The sentence to carry forward: **every number the twin will later reason about is a tuple this chapter's components produced, and a large part of whether the twin deserves to be believed is settled before any model runs.**

Exercises

Solutions or hints follow the exercises; each says which. The objective each exercise serves is given in brackets. None of the exercises uses the demonstrator.

Exercise 1 (Objective 1, easy). A datasheet for a pressure sensor gives: range 0 to 10 bar, resolution 0.01 bar, accuracy plus or minus 0.1 bar, drift 0.05 bar per year, response time 50 ms, maximum sample rate 100 Hz. A colleague proposes an alarm when the pressure changes by more than 0.05 bar between consecutive readings. Say what is wrong with the proposal, and what the smallest defensible threshold is.

Exercise 2 (Objective 2, easy). A cold-storage room's compressor cycles every 20 minutes. Its temperature is sampled every 20 minutes. Say what the record will show, then propose two different sampling intervals that would show the true cycle, and say which of the two you would use and why.

Exercise 3 (Objective 2, medium). A twelve-bit converter reads a sensor that outputs 0 to 5 V over a level of 0 to 2 metres of water. Compute the quantisation step in metres and the worst-case rounding error. Then say whether a compression bound of 1 mm would lose anything the sensor had.

Exercise 4 (Objective 4, medium). A gateway's clock is 3 seconds fast. For each of three bindings – a room temperature every 5 minutes, a motor current every 100 ms, and a power-quality waveform at 10 kHz – compute the clock error as a fraction of the sampling interval and say whether network time synchronisation is enough.

Exercise 5 (Objective 3, medium). Score each transport against the five properties, giving a yes, a no, or "by convention, if the payload carries X": (a) a vendor's REST endpoint that returns only the latest value; (b) bare MQTT with quality-of-service level 1; (c) MQTT whose publisher puts a sequence number and an event time in every payload and registers a will message; (d) a nightly CSV file named by date; (e) a Kafka topic written by the gateway.

Exercise 6 (Objective 4, medium). A twin's state document shows `as_of` of 06:10:00 and the wall clock reads 06:47:30. The binding's sampling interval is 10 minutes and the configuration's staleness bound for issuing a command is 15 minutes. Compute the age of twin, say what the lifecycle field should read, and say what a command-path check does.

Exercise 7 (Objectives 5 and 7, medium). For each data shape, choose an encoding in flight and a store and layout at rest, and say what each preserves and loses: (a) forty temperatures per second from a kiln over a metered satellite link; (b) a maintenance report typed by a technician; (c) a thermal camera's frames once a minute; (d) the record that says which sensor sits on which pump of a pumping station, with its register, scale, unit and calibration date.

Exercise 8 (Objective 6, medium). Write, in the shape of the validate listing, a clean function for a binding whose readings arrive over MQTT with a sequence number in the payload. It must detect a gap, a duplicate and an out-of-order tuple, and say what quality

state, if any, it attaches in each case. Then write the three tests, each a tuple and an expected outcome.

Exercise 9 (Objective 7, medium). A fleet of 300 pumps each has 12 sensors sampled every second, with 16 bytes per point. Compute the raw volume per day for the fleet. Then propose a retention and downsampling policy with three tiers, say which summaries the second tier keeps and why, and compute the volume of the third tier after a year.

Exercise 10 (Objective 7, medium). A colleague proposes a time-series schema with tags for site, pump, sensor, firmware version, and the sequence number of each message. Say which tag is the mistake, why it is hard to reverse, and where that value belongs instead.

Exercise 11 (Objectives 3 and 4, hard). Finish Example 2 of the Faded Examples section: the late-data policy for the nightly state-of-health computation, what the store must keep for it, and the age of twin at midnight for the bus that was out of coverage.

Exercise 12 (Objectives 1 to 7, hard, open-ended). Do Example 3 of the Faded Examples section, the district heating network, in all eight parts.

Solutions and hints

Exercise 1 (solution). The threshold is below the sensor's accuracy (0.1 bar) and at half its drift per year; a change of 0.05 bar between readings is inside the sensor's own uncertainty and the alarm fires on the instrument. The smallest defensible threshold is above the accuracy – a change of more than 0.1 bar, and in practice a multiple of it – and it should be a change sustained over several readings, since response time and noise make a single-step test unreliable.

Exercise 2 (solution). The record shows a nearly constant temperature, or a slow phantom drift as the sample instant walks round the compressor's cycle – aliasing. Any interval well under half the cycle shows the true cycle: every 2 minutes gives ten samples per cycle; every 5 minutes gives four. Use 2 minutes if the decision is about the compressor's health (the shape of the cycle matters) and 5 minutes if the decision is about the room's mean temperature (an average over the cycle is enough); either way run the check – sample every 10 seconds for a day and decimate – before trusting the choice.

Exercise 3 (solution). $2^{12} = 4096$ counts across 5 V is 1.22 mV per count; across 2 metres it is $2/4096 = 0.00049$ metres, about half a millimetre per count, and the worst-case rounding error is a quarter of a millimetre. A compression bound of 1 mm is two counts, larger than the quantisation step, so it loses something the sensor had – a small something, and defensible, but not free. A bound of 0.2 mm would be free.

Exercise 4 (solution). Room temperature: $3/300 = 0.01$, one per cent of the interval; network time is ample. Motor current: $3/0.1 = 30$ samples displaced; network time, which holds tens of milliseconds, is enough for the interval but the 3-second error itself is not acceptable and says the gateway is not synchronised at all. Waveform: $3/0.0001 = 30{,}000$ samples displaced; network time cannot serve this binding and a precision time source at the gateway is required.

Exercise 5 (hint). (a) None of the five for free; event time only if the payload carries it; not resumable, so a gap is unrecoverable. (b) Delivery at least once gives duplicates with no way to tell them apart unless the payload carries a sequence number; connection state only via

the will; no gap recovery. (c) Four by convention – the will message for state, the event time for time, the sequence number for ordering, duplicates and the visibility of a gap – and gap recovery only if the publisher also keeps a buffer it can be asked to resend from. (d) Gap survivable and ordered by construction; connection state unobservable until the file fails to appear; event time preserved if the rows carry it. (e) Ordering, deduplication and gap recovery from the log's offsets; event time and connection state still depend on what the gateway wrote.

Exercise 6 (solution). The age of twin is 37 minutes 30 seconds. Against a 10-minute interval that is nearly four missed updates, so the lifecycle field should read *out of sync*; against a 15-minute staleness bound the command path must refuse, hold the last safe command, and raise the inward alert – Chapter 3's staleness row, computed from Chapter 4's `as_of` by this chapter's Equation 3.

Exercise 7 (hint). (a) A binary schema-required encoding with a sequence number, compressed at the edge within the sensor's resolution; columnar at rest; loses nothing the sensor had, costs a versioned schema. (b) Self-describing text, since a person wrote it and a person will read it; a document store; loses nothing, gains nothing from compression. (c) The frame as a file with a JSON sidecar; an object store; the tuple's value is the object key. (d) Metadata: a self-describing JSON record per binding with a `valid_from`, in a document store or a table keyed by binding, versioned rather than overwritten; loses nothing, is read rarely, and is what makes every other store's numbers mean something.

Exercise 8 (hint). Keep the last sequence number and last event time per binding. A sequence number more than one ahead is a gap: pass the tuple, record the gap, and mark the window's count short. The same sequence number twice is a duplicate: drop the second, attach nothing. An event time earlier than the last stored is out of order: buffer it for the grace period and insert it in order, or emit a correction if the window has closed. Only a filled value gets `substituted`; the other two attach no quality state, because the tuple itself is good – it is the stream that had the defect.

Exercise 9 (solution). $300 \times 12 \times 86{,}400 \times 16 = 4.98 \times 10^9$ bytes a day, about 4.6 GiB for the fleet. Three tiers: raw for 30 days; one-minute windows with mean, min, max and count for two years; hourly windows with the same four for the asset's life. The second tier keeps min and max so a spike survives and count so a gap is visible. The third tier after a year: $300 \times 12 \times 8760$ windows of four 8-byte values plus a timestamp, about $3.15 \times 10^7 \times 40 = 1.26 \times 10^9$ bytes, a little over a gigabyte – four thousand times smaller than the raw year.

Exercise 10 (solution). The sequence number: its distinct values grow with every message, so the index grows with the history rather than with the fleet, and removing it later means rewriting every point. It belongs as a field, or better in the clean function's own state, since its only job is gap and duplicate detection. Firmware version is a legitimate tag – a handful of values per fleet – and the digital thread wants it there.

Exercise 11 (solution). The nightly computation must emit corrections: it runs at midnight over the day's data, and when the missing hour arrives at 01:00 it recomputes that day's state of health and publishes it again, marked as superseding the first. For that to be possible the store must keep the raw cell data for at least the day, not only the nightly result, and the result must carry the version and the data span it was computed from. At midnight that bus's age of twin should have read about one hour and its lifecycle *out of sync*, and a

dashboard showing its state of health with that age beside it is telling the truth.

Exercise 12 (open). Assessed on whether the three sources reach one tuple with three different quality vocabularies rather than one; on whether the quarterly readings are treated as events with a stated uncertainty rather than as samples; on whether the MQTT clock drift is detected from the time discrepancy and impossible timestamps are quarantined rather than corrected; and on whether the dashboard for the oldest substations shows an age of twin of three months rather than a number.

Further Reading

- *Sensing and sampling:* [5] for a published twin whose sensor array is specified with ranges and accuracies, the real form of the datasheet table; [6] for calibration and wear as sources of bad data reaching a twin; [9] for the distinction between a sensor's sampling frequency and the twinning rate. The sampling theorem, aliasing and quantisation are textbook material and the mathematics appendix has the argument.
- *Time and synchronisation:* [7] for time discrepancy between digital and physical twins studied in its own right; [8] for timeliness, reliability and consistency metrics as indicators that something is wrong; [10] for twinning rate and age of twin defined as platform metrics.
- *Transports and industrial protocols:* [11] and [12] for HTTP and MQTT side by side and for MQTT's broker features; [13] and [14] for MODBUS and OPC UA as twins actually use them; [15], [16] and [9] for OPC UA's information model, address space and two messaging patterns; [17] for Kafka against other publish/subscribe systems; [18] for one handler per protocol at scale; [1] for heterogeneity as middleware's problem. The protocol samples are this book's own construction.
- *Encodings:* [10] for the encoding family and payload size as latency; [19] for OPC UA PubSub with a binary and a JSON encoding over UDP and MQTT; [20] for the UNECE unit codes as a code list. The six encodings of one reading and their byte counts are this book's own construction.
- *Functions:* [21] for the properties of a twin – reflection, entanglement, memorisation, composability, augmentation and predictability – that the platform literature builds on; [22] and [23] for the shadowing function computing state and propagating actions; [18] for augmentation and composition as the capabilities a passive twin forgoes; [24] for edge functions in the Industrial Internet Reference Architecture; [25] for function as a service.
- *Stream processing and placement:* [2] for hot, warm and cold paths and Kafka as the ingest stream; [22] and [27] for functions relocated along the continuum; [26] for component placement measured on a testbed; [28] for when offloading to the cloud wins.
- *Storage:* [30], [31] and [29] for model-based compression within an error bound, the turbine volume and bandwidth figures, Parquet and Arrow in an edge-to-cloud pipeline; [35] for data lakes that keep raw data; [32], [33], [34], [9] and [2] for NetCDF, labelled arrays, video arrays with sidecar metadata, point clouds and images on a topic. The metadata record is this book's own construction.
- *Data quality:* [8] for the experimental form of the gaps, duplicates and out-of-order table; [6] for the sensor-side causes. The six quality states are this book's own construction.

- *The lifecycle view:* [3] for the digital thread as the Digital Twin Consortium defines it and [4] for the twin as shadow of the present and thread of the past.
- *The faded examples' sources:* [38] and [39] for what a turbine's SCADA system reports and at what rate; [40] for what a battery management system measures per cell and how often.

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] 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.

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

[4] R. Saracco, "Digital Twins: Bridging Physical Space and Cyberspace," *Computer*, vol. 52, no. 12, pp. 58–64, Dec. 2019, doi: 10.1109/MC.2019.2942803.

[5] P.-E. Goffi, R. Tremblay, and B. Oakes, "Engineering a Digital Twin for the Monitoring and Control of Beer Fermentation Sampling." arXiv, Aug. 2025. doi: 10.48550/arXiv.2508.18452.

[6] C. Savaglio, V. Barbuto, F. Mangione, and G. Fortino, "Generative Digital Twins: A Novel Approach in the IoT Edge-Cloud Continuum," *IEEE Internet of Things Magazine*, vol. 8, no. 1, pp. 42–48, Jan. 2025, doi: 10.1109/IOTM.001.2400035.

[7] 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.

[8] M. Picone, E. Bouquillon, M. Pietri, and M. Mamei, "Assessing the Impact of Cybersecurity Attacks on Digital Twin Metrics: An Experimental Study," in *2026 IEEE 23rd Consumer Communications & Networking Conference (CCNC)*, IEEE, 2026, pp. 1–4. Accessed: Mar. 27, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11366535/

[9] T. Böttjer *et al.*, "A review of unit level digital twin applications in the manufacturing industry," *CIRP Journal of Manufacturing Science and Technology*, vol. 45, pp. 162–189, Oct. 2023, doi: 10.1016/j.cirpj.2023.06.011.

[10] 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.

[11] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *2017 IEEE international systems engineering symposium (ISSE)*, IEEE, 2017, pp. 1–7. Accessed: Feb. 25, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8088251/

[12] B. Mishra and A. Kertesz, "The use of MQTT in M2M and IoT systems: A survey," *Ieee Access*, vol. 8, pp. 201071–201086, 2020, Accessed: Feb. 25, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9247996/

[13] A. R. Qureshi, A. Asensio, M. Imran, J. Garcia, and X. Masip-Bruin, "A survey on security enhancing Digital Twins: Models, applications and tools," *Computer Communications*, vol. 238, p. 108158, Jun. 2025, doi: 10.1016/j.comcom.2025.108158.

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

[15] S. Cavalieri and S. Gambadoro, "Proposal of Mapping Digital Twins Definition Language to Open Platform Communications Unified Architecture," *Sensors*, vol. 23, p. 2349, Feb. 2023, doi: 10.3390/s23042349.

[16] Q.-D. Nguyen, Y. Huang, F. Keith, C. Leroy, M.-T. Thi, and S. Dhouib, "Manufacturing 4.0: Checking the Feasibility of a Work Cell Using Asset Administration Shell and Physics-Based Three-Dimensional Digital Twins," *Machines*, vol. 12, p. 95, Jan. 2024, doi: 10.3390/machines12020095.

[17] 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.

[18] M. Picone, M. Mamei, and F. Zambonelli, "A Flexible and Modular Architecture for Edge Digital Twin: Implementation and Evaluation," *ACM Transactions on Internet of Things*, vol. 4, no. 1, pp. 8:1–8:32, Feb. 2023, doi: 10.1145/3573206.

[19] "Summary of IoT, and DT Standards." Change2Twin project.

[20] "AutomationML: A Practical Guide," in *AutomationML*, De Gruyter Oldenbourg, 2021. doi: 10.1515/9783110746235.

[21] R. Minerva, G. M. Lee, and N. Crespi, "Digital twin in the IoT context: A survey on technical features, scenarios, and architectural models," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1785–1824, 2020, Accessed: Feb. 25, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9120192/

[22] L. Bedogni, M. Mamei, M. Picone, M. Pietri, and F. Zambonelli, "Fluid Computing & Digital Twins for intelligent interoperability in the IoT ecosystem," *Future Generation Computer Systems*, vol. 171, p. 107855, Oct. 2025, doi: 10.1016/j.future.2025.107855.

[23] A. Ricci, A. Croatti, S. Mariani, S. Montagna, and M. Picone, "Web of Digital Twins," *ACM Transactions on Internet Technology*, vol. 22, no. 4, pp. 1–30, Nov. 2022, doi: 10.1145/3507909.

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

[25] S. Deng *et al.*, "Cloud-Native Computing: A Survey From the Perspective of Services," *Proceedings of the IEEE*, vol. 112, no. 1, pp. 12–46, Jan. 2024, doi: 10.1109/JPROC.2024.3353855.

[26] S. M. Raza, R. Minerva, B. Martini, and N. Crespi, "Empowering Microservices: A Deep Dive into Intelligent Application Component Placement for Optimal Response Time," *Journal of Network and Systems Management*, vol. 32, no. 4, p. 84, Aug. 2024, doi: 10.1007/s10922-024-09855-3.

[27] P. Bellavista, N. Bicocchi, M. Fogli, C. Giannelli, M. Mamei, and M. Picone, "Exploiting microservices and serverless for Digital Twins in the cloud-to-edge continuum," *Future Generation Computer Systems*, vol. 157, pp. 275–287, Aug. 2024, doi: 10.1016/j.future.2024.03.052.

[28] R. Singh, J. Kovacs, and T. Kiss, "To Offload or Not? An Analysis of Big Data Offloading Strategies from Edge to Cloud," in *2022 IEEE World AI IoT Congress (AIIoT)*, Jun. 2022, pp. 046–052. doi: 10.1109/AIIoT54504.2022.9817276.

[29] S. K. Jensen, C. S. Godiksen, C. Thomsen, and T. B. Pedersen, "Demonstration of ModelarDB: Model-Based Management of High-Frequency Time Series Across Edge, Cloud, and Client," *Proceedings of the VLDB Endowment*, vol. 18, no. 12, pp. 5247–5250, Aug. 2025, doi: 10.14778/3750601.3750643.

[30] S. K. Jensen, C. Thomsen, and T. B. Pedersen, "ModelarDB: Integrated Model-Based Management of Time Series from Edge to Cloud," in *Transactions on Large-Scale Data- and Knowledge-Centered Systems LIII*, A. Hameurlain and A. M. Tjoa, Eds., Berlin, Heidelberg: Springer, 2023, pp. 1–33. doi: 10.1007/978-3-662-66863-4_1.

[31] A. Abduvakhobov, S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Scalable Model-Based Management of Massive High Frequency Wind Turbine Data with ModelarDB," *Proceedings of the VLDB Endowment*, vol. 17, no. 13, pp. 4723–4732, Sep. 2024, doi: 10.14778/3704965.3704978.

[32] Y. Ye, A. González-Vidal, A. Cisterna-García, A. Pérez-Ruzafa, M. A. Z. Izquierdo, and A. F. Skarmeta, "Advancing Towards a Marine Digital Twin Platform: Modeling the Mar Menor Coastal Lagoon Ecosystem in the South Western Mediterranean." arXiv, Sep. 2024. doi: 10.48550/arXiv.2409.10134.

[33] J. Vaillant, I. Grechi, F. Normand, and F. Boudon, "Towards virtual modelling environments for functional–structural plant models based on Jupyter notebooks: Application to the modelling of mango tree growth and development," *in silico Plants*, vol. 4, no. 1, p. diab040, Jan. 2022, doi: 10.1093/insilicoplants/diab040.

[34] J. Slavič *et al.*, "Python open-source signal processing and modeling/simulation for scientific research in structural dynamics—A review," *Mechanical Systems and Signal Processing*, vol. 241, p. 113465, Dec. 2025, doi: 10.1016/j.ymssp.2025.113465.

[35] J. Chae, S. Lee, J. Jang, S. Hong, and K.-J. Park, "A Survey and Perspective on Industrial Cyber-Physical Systems (ICPS): From ICPS to AI-Augmented ICPS," *IEEE Transactions on Industrial Cyber-Physical Systems*, vol. 1, pp. 257–272, 2023, doi: 10.1109/TICPS.2023.3323600.

[36] C. Badolato *et al.*, "PSMark: A distributed IoT benchmark for publish/subscribe under domain-based workloads," in *PerCom 2026-24th IEEE International Conference on Pervasive Computing and Communications*, 2026. Accessed: Mar. 18, 2026. [Online]. Available: https://inria.hal.science/hal-05517145/

[37] M. Iorio, F. Risso, and C. Casetti, "When latency matters: Measurements and lessons learned," *SIGCOMM Comput. Commun. Rev.*, vol. 51, no. 4, pp. 2–13, Dec. 2021, doi: 10.1145/3503954.3503956.

[38] 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.

[39] L. A. Bull *et al.*, "Data-Centric Monitoring of Wind Farms: Combining Sources of Information," in *Data Driven Methods for Civil Structural Health Monitoring and Resilience*, CRC Press, 2023.

[40] A. Thelen, X. Huan, N. Paulson, S. Onori, Z. Hu, and C. Hu, "Probabilistic machine learning for battery health diagnostics and prognostics—review and perspectives," *npj Materials Sustainability*, vol. 2, no. 1, pp. 1–33, Jun. 2024, doi: 10.1038/s44296-024-00011-1.