Chapter 14 – Running Twins in Production: Deployment, Evolution, and the Twin's Own Lifecycle

14.0 Before you start

Where we are. Two chapters of this book say, in identical words, "Chapter 14 is the one this chapter leans on hardest". Chapter 7 said it about its expiry conditions and Chapter 8 said it about its retraining triggers, and Chapter 7 put the omission precisely:

> **Expiry conditions are only worth writing if something acts on them.**

Nothing acts on them. Across Chapters 7 to 13 this book has written **twenty-four** conditions under which something a twin depends on stops being true, distributed across seven chapters and six design documents, and **no chapter built the thing that fires them.** Sec. 14.1 collects them, and the collection is the chapter's opening artifact.

The register, unchanged from Chapters 9 to 13. You have operated software in production. You know what a runbook is, what an on-call rotation costs, and what a postmortem is for.

> **This chapter does not teach operations. It teaches what is different about operating a system whose correctness decays because the world changed, rather than because somebody deployed something.**

That sentence is the whole distinction. Ordinary software is wrong when you change it. A twin is wrong when the *pot is repotted*, and nobody deployed anything.

What you are assumed to know. Everything so far. Especially: Chapter 1's cost and payback estimate in Sec. 1.8.7 and its falsifier in Sec. 1.8.8; Chapter 3's registry, its binding rule and its lifecycle concern; Chapter 5's reproducibility requirement and its simulate-the-physical-twin arrangement; Chapter 7's credibility argument, its five expiry triggers and its three monitors; Chapter 8's three retraining triggers and the retroactive one; Chapter 9's sensor drift and quality states; Chapter 10's validity intervals, dataset definitions and provenance chain; Chapter 11's three outcomes and work queue; Chapter 12's cost model and expiry triggers; Chapter 13's counterparty rule.

The maths budget. As Chapters 9 to 13. One set piece, and it closes a loop the book opened in Chapter 1: **the operating bill** (Sec. 14.4). Engineering days are counted separately from lab-operations days throughout, because walking to a rig is not the twin's operating cost and mixing the two inflates the answer.

What this chapter deliberately does not cover. Operations practice – runbooks, on-call design, incident-management frameworks. Continuous Integration and Continuous Delivery (CD) mechanics and container technology – TwinOps and the DevOps-for-twins work are named as the shape, and Chapter 12's refusal to name technology holds here. MLOps as a technology – Chapter 8 deferred *what is different when the model is learned*, which is Sec. 14.3.4. Security in general – Chapter 3 Sec. 3.3.3 owns it and said Chapter 14 "returns to running this in production", so Sec. 14.6.4 covers what changes operationally and defers the rest. Fleet operations and anything that only appears at many twins – Chapter 15; Sec. 14.4's arithmetic is deliberately for one.

Learning objectives

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

1. **Assemble** an expiry register from the conditions a twin's design documents already contain, and **classify** each trigger by what can detect it.
2. **Design** a release process for a twin, including what stands in for the staging environment a physical twin does not have.
3. **Compute** a twin's annual operating cost from its own design, and **recompute** the payback and the falsification threshold from Chapter 1.
4. **Distinguish** a twin whose parts have changed from one whose purpose has changed, and **explain** why the second is harder to detect.
5. **Decide** what to automate first, and **justify** it from the operating bill rather than from preference.

14.1 The expiry register

14.1.1 Twenty-four conditions, seven chapters, nobody watching

Here is every expiry condition this book has written, collected for the first time. Read it as the indictment it is.

From	The condition
Ch7 Sec. 7.7.2	4 weeks since the last calibration
Ch7 Sec. 7.7.2	The physical twin changed: repotted, plant replaced, dripper moved
Ch7 Sec. 7.7.2	The model version changed
Ch7 Sec. 7.7.2	The 14-day rolling residual bias left plus or minus 1.0
Ch7 Sec. 7.7.2	A sensor was replaced or recalibrated
Ch7 Sec. 7.7.4	The residual spread grew materially
Ch7 Sec. 7.7.4	The estimator gain moved outside its stated band
Ch8 Sec. 8.6.3	The model was retrained
Ch8 Sec. 8.6.3	The input-coverage refusal rate exceeded its threshold
Ch8 Sec. 8.6.3	The training data's certification as "normal" was called into question – **and this one invalidates backwards**
Ch9 Sec. 9.2.2	A sensor's specified drift has consumed its calibration margin
Ch9 Sec. 9.3.3	Something changed near the sensor, so the aliasing check is stale
Ch10 Sec. 10.3.5	The retention policy is about to delete something a claim depends on
Ch11 Sec. 11.9.2	The alert threshold or the confirmation rule changed
Ch11 Sec. 11.8.3	The evaluation work queue is backing up

From	The condition
Ch12 Sec. 12.8.4	A second twin is funded
Ch12 Sec. 12.8.4	Data volume rose by two orders of magnitude
Ch12 Sec. 12.8.4	Somebody wants a service that was declined
Ch12 Sec. 12.8.4	**The person who owns the shared parts left**
Ch13 Sec. 13.6.5	A modelling group is contracted; a second bench appears; equipment speaks OPC UA; an external reviewer arrives; a learned model reaches production

Twenty rows, and **twenty-four conditions** – Chapter 13's five share a row because they were written as one list. Seven chapters, six design documents, **each of which ended by saying somebody should watch these.**

That is a normal outcome and not a failure of diligence. Expiry conditions are written at the end of a design activity by the person who has just finished designing, and they are read by nobody, because there is no document whose job it is to hold them.

14.1.2 What the register is

One list. One owner. One review cadence.

```
expiry_register(
    id, condition,              -- in words a non-author can check
    source,                     -- which document asserted it
    detector,                   -- what will notice: a monitor, a date, a person
    owner,                      -- a named role
    last_checked, status,
    on_fire                     -- what happens: refit, re-validate, re-decide, escal
)
```

Four properties make it work, and each is cheap only if done at the start.

It is one list, not a section in six documents. The value is entirely in the aggregation: a reviewer reading twenty-four conditions in one place asks "which of these has no detector?", and that question cannot be asked of six documents.

Every row has a detector. Sec. 14.1.3 classifies them, and the classification is the point of the exercise.

Every row has an owner who is a role, not a person. Chapter 12 Sec. 12.8.4's fourth trigger – the person who owns the shared parts leaves – is the register auditing itself, and it only works if rows name roles.

It is reviewed on a cadence, and the review is short. Monthly, fifteen minutes, three questions: has anything fired, has anything become undetectable, and has anything been added by a design decision since last month.

14.1.3 Three kinds of trigger, and only one fires by itself

Classify every row. The classification decides where your effort goes.

Kind	Detected by	Examples from Sec. 14.1.1	Failure mode
Self-firing	A monitor, from data the twin already has	Residual bias, residual spread, estimator gain, coverage refusal rate, queue depth	Alert fatigue, or a monitor nobody wired to anything
Scheduled	A calendar	4 weeks since calibration; sensor drift margin; the aliasing re-check	Quiet omission – nobody notices a job that stopped running
Announced	A human telling you	Repotting, sensor replacement, a contracted modelling group, a second twin, an external reviewer, the departure of the parts owner	**Silence is indistinguishable from nothing having happened**

Count the rows. Of the twenty-four, seven are self-firing, four are scheduled, and **thirteen are announced**. More than half of this book's expiry conditions depend on somebody remembering to say something.

Figure 14.1 is that count drawn to scale, because the ratio is the finding and a table hides proportion.

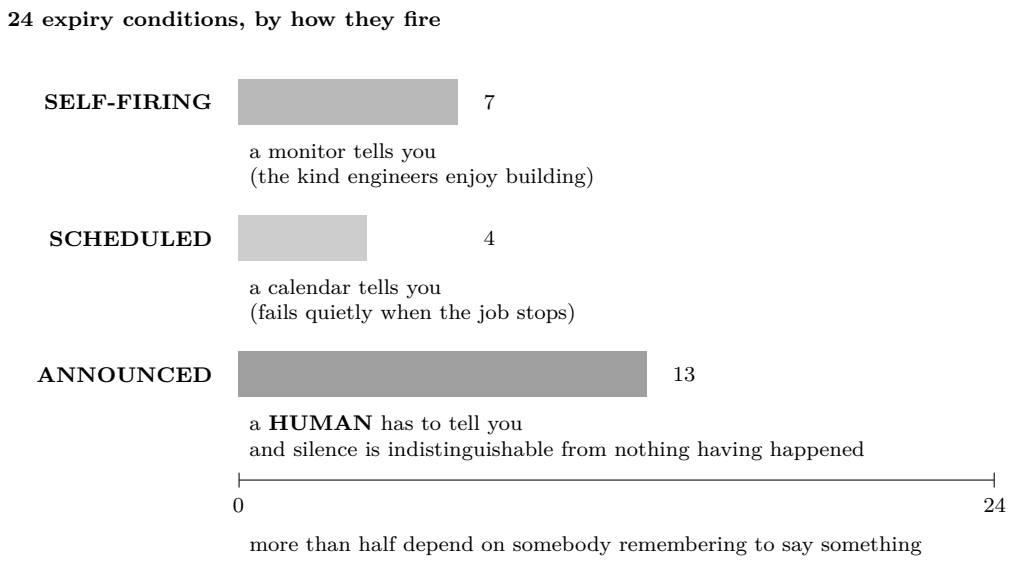


Figure 14.1 Where the expiry conditions actually live. Effort tends to go to the left bar; the risk is in the right one.

That ratio is the chapter's first finding, and it should change what you build. The self-firing triggers are the ones engineers enjoy building and they are a minority. The majority need a *process*, and a process without an artifact is a hope. Sec. 14.3.5 is about whether the announced category has an answer at all.

14.2 Deployment: what is different about shipping a twin

14.2.1 You are shipping four things, not one

An ordinary release ships code. A twin release ships up to four independently-versioned things, and they have different risks and different gates.

Artifact	Changes when	Gate
Code	Ordinary development	Your existing tests
Models	A refit, a structural change, a retrain	Chapter 7's hold-out; Chapter 7's golden trajectories
Parameters	Every recalibration – Chapter 7's four-week cadence	The hold-out, and a diff a human reads
Configuration and bindings	The physical twin changes	Chapter 3's rule that binding is data; Chapter 10's validity intervals

Two consequences that catch people.

Three of the four change without anybody writing code, so a release process gated only on code review has no gate on three quarters of what ships. Chapter 3 Sec. 3.2.4 said the registry "is where that cost is either managed or lost", and this is that sentence in operational form.

A parameter change is a release. Chapter 7's recalibration produces three numbers, and those three numbers change what the twin alerts on. Treating them as data rather than as a release is how a twin's behaviour changes with no record of a change – which Chapter 10's whole provenance chain exists to prevent, and which the deployment process can undo in one step.

14.2.2 The physical twin has no staging environment

Here is the structural difference, and everything in this section follows from it.

You cannot spin up a second greenhouse. You cannot restart the turbine to test a release. **The system your software talks to is a single, expensive, slow, unrepeatable instance of the world**, and it is the only one. Figure 14.2 shows what that removes from a pipeline you already know.

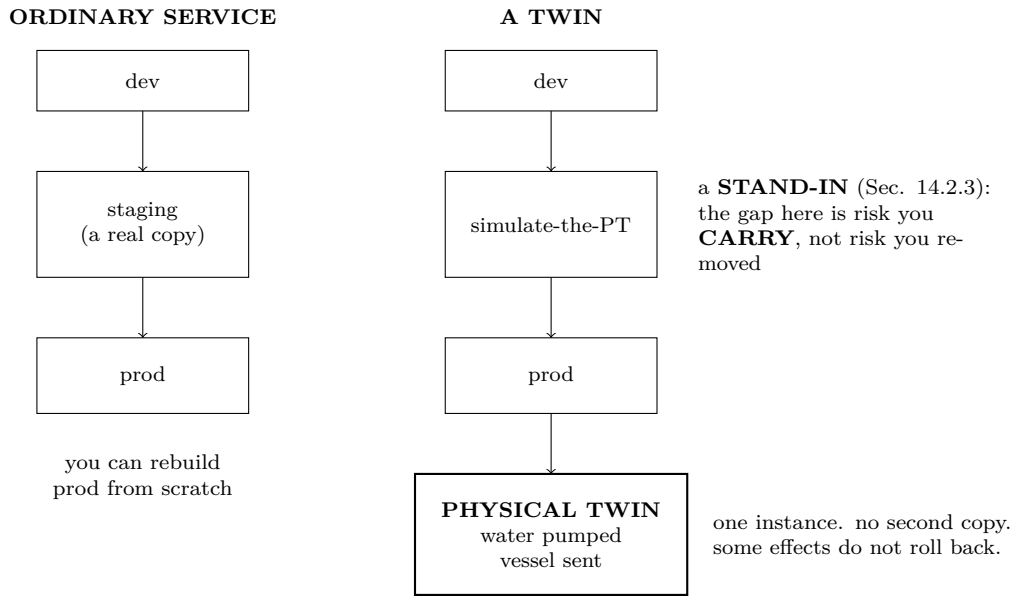


Figure 14.2 The missing box. Every deployment practice in this section is a way of paying for a staging environment that cannot exist.

Three consequences.

You cannot test the ingest path end to end without the physical twin. Whatever you test against is a stand-in, and the difference between the stand-in and the world is a risk you carry rather than eliminate.

Failures are not reproducible on demand. You cannot make the pump fail to see what happens. Chapter 7 Sec. 7.5.6 already said the corresponding thing about validation: you cannot validate a prediction about a failure that has never happened.

The blast radius of a bad release includes physical things – at Stage 3, water; on a turbine, a vessel dispatched or not dispatched. Chapter 2 Sec. 2.8's boundary arrives at deployment time as well as at design time.

14.2.3 Simulate-the-physical-twin, which is what stands in

Chapter 5 Sec. 5.5.3 built this and Chapter 5 Sec. 5.12 sent it here as a practice. It is the answer to Sec. 14.2.2.

What it is. Run the *same model* forward to produce synthetic measurements, and point the twin at those instead of at the physical twin. Chapter 5 already noted this is the same machinery – a model, an initial state, inputs, a horizon and a step size – so it costs almost nothing once the runner exists.

What it buys, and it is more than a test environment.

A pipeline test with known answers. You know what the moisture *should* read because you generated it, so every stage from connector to alert can be checked against ground truth. This is the only place in a twin's life where ground truth exists.

Failures on demand. Generate a dose that delivers nothing and confirm the alert fires. Generate a stuck sensor – Chapter 9 Sec. 9.7.4's undetectable case – and confirm

what happens. **This is where Chapter 11's three outcomes get tested**, because `cannot_evaluate` is otherwise only exercised by real infrastructure failures you have to wait for.

A test that runs in seconds. Chapter 5 Sec. 5.3.4 costed a run at 32 microseconds, so a simulated week is instant.

And the honest limit, which must be stated wherever this is used. A simulated physical twin generated by the same model the twin uses cannot find a modelling error, because it shares the error. It tests the *plumbing*, not the *physics*. Chapter 7's hold-out is the only thing that tests the physics, and no amount of synthetic data substitutes for it.

> **So the split is clean and worth memorising.** Simulate-the-physical-twin tests everything from the connector to the alert, with ground truth, in seconds. Chapter 7's hold-out tests whether the model deserves belief. A release process needs both and they answer different questions.

14.2.4 Golden trajectories as the release gate

Chapter 7 Sec. 7.3.3 built golden trajectories as a verification device and described three uses. In production they become the gate:

> **No model, parameter or configuration change reaches the twin until it has been run against the recorded scenarios and the diff has been read by a human.**

Why a human reads the diff rather than a threshold passing it. A model change is *supposed* to change the outputs. An automatic pass/fail on "outputs unchanged" would reject every legitimate change and accept nothing useful. What the gate produces is a sentence: "version 3 predicts evening readings 4 units lower on average across the twelve regression scenarios, and differs by more than 10 units only in the saturation scenario." A human decides whether that sentence is the change they intended.

Four scenarios are the minimum, and the demonstrator's are already specified in Chapter 7 Sec. 7.8.2: a dry-down, a normal day, a missed evening dose, and a saturation excursion. Add one per incident, forever – **an incident that does not leave a scenario behind will happen again.**

14.2.5 Shadow deployment, rollback, and the thing you cannot roll back

Shadow deployment is the twin-specific pattern worth naming. Run the new version alongside the old, against the same inputs, and **do not act on the new one**. Compare for a period, then promote. For a twin this is unusually cheap – the models are small, the runner is fast, and Chapter 10's schema already stores derived output keyed on `(binding, as_of, decision_time)`, so two versions producing beliefs about the same moment is a supported case rather than a conflict.

Run a shadow for at least one full cycle of the thing you care about. For the demonstrator that is a week, because Chapter 7's residual behaviour is weekly and a shadow of two days would miss the day-night structure that Chapter 7 Sec. 7.4.6 earned a parameter for.

Rollback works for three of the four artifacts. Code, models and parameters are versioned and revertible; Chapter 10's registry holds the previous ones and Chapter 8 Sec. 8.2.3's content hashes prove which is which. Reverting is a normal operation.

What you cannot roll back is what the twin already did. An alert that was sent, a schedule that was changed, water that was delivered. Chapter 2 Sec. 2.8's asymmetry – a shadow that is wrong displays something false, a twin that is wrong *does* something false – is exactly the asymmetry between a revertible and an unrevertible release.

> **Which gives the deployment rule for a closed-loop twin, and it is short.** The gate before the command path is not the same gate as the one before a dashboard. Ship the advisory version, shadow the acting version against it, and let a human promote – **and note that Chapter 12 Sec. 12.13 flagged the version bump that changes a number as the exact event this gate exists for.**

14.3 The operating loop

14.3.1 Five recurring activities

Everything the twin needs done, repeatedly, once it is running:

Activity	Cadence	Triggered by
Watch – monitors, digests, queue depth	Continuous, reviewed weekly	Ch7's three monitors, Ch11's daily digest, Ch11 Sec. 11.8.3's queue
Recalibrate – refit parameters	Ch7's four weeks, or a fired trigger	Scheduled and self-firing
Re-validate – re-run the hold-out, re-issue the argument	After every recalibration or model change	Follows recalibration
Record – enter physical changes, resolutions, decisions	On the day it happens	Announced (Sec. 14.1.3), and this is the weak one
Review – the expiry register	Monthly, fifteen minutes	Calendar

Notice that four of the five are cheap and one is not. Recalibration and its re-validation are where the hours go, and Sec. 14.4 counts them.

14.3.2 Recalibration as a workflow

Chapter 7 Sec. 7.4 taught the technique. Here it is as something that happens thirteen times a year without a person deciding to do it each time.

```
1. Materialise the calibration dataset      Ch10 Sec. 10.7.2: a stored query,
                                            an ingest watermark, an exclusion list
2. Refuse if the windows are degraded       Ch9 Sec. 9.7.1: completeness below 90%
```

```
3. Fit                                        Ch7 Sec. 7.4.1: value plus spread, per p
4. Compare against the current values         A diff a human reads: three numbers
5. Run the hold-out on a fresh window         Ch7 Sec. 7.4.5: bias and spread
6. Run the golden trajectories                Sec. 14.2.4: the release gate
7. Register the new parameter set             Ch10 Sec. 10.4.3: a new row, valid_from
8. Re-issue the credibility argument          Ch7 Sec. 7.7.1: five parts, updated
9. Reset the register row                     Sec. 14.1.2: last_checked, status
```

Nine steps, and eight of them are mechanisable. Step 4 is the one that needs a human, and it should stay that way: **a parameter that moved a lot is information, and a pipeline that silently accepts it has discarded the signal.** Chapter 7 Sec. 7.4.7 asked for a parameter history precisely so that "the twin has been getting worse for a month" becomes a one-line diff.

Two operational rules that are cheap now and expensive later.

Never recalibrate and change the model in the same release. If both move, the diff at step 4 is uninterpretable and you have lost the only cheap check you had.

A refusal at step 2 is a finding, not an error. Degraded windows mean the connector had a bad month, which is a Chapter 9 problem the recalibration job has just discovered for free. Route it, do not retry it.

14.3.3 Re-validation, and the cadence problem

Chapter 7 Sec. 7.5.6 named the problem and Chapter 8 Sec. 8.5.2 sharpened it: *if your validation procedure takes three weeks and your model updates every week, you do not have a validation procedure – you have a backlog.* Chapter 8 said Chapter 14 would solve it operationally.

The operational solution is not to make validation faster. It is to recognise that there are two different validations and they have different cadences.

	Full re-validation	Continuous validation
What it is	Chapter 7's whole procedure: fresh hold-out, convergence check, envelope review, argument re-issued	Chapter 11 Sec. 11.5.4's scoring loop: every prediction scored against what happened
Cadence	Quarterly, or on a structural change	Continuous
Cost	Days	Zero, once built
Catches	Everything, including envelope and assumption problems	Drift, bias, and a widening spread
Cannot catch	Nothing, but it is slow	A model that is wrong in a way the current conditions do not exercise

The cadence problem dissolves once you stop treating validation as one activity. The continuous half runs at the speed of the data and is

what licenses a faster refit cadence; the full half runs quarterly and is what the credibility argument is re-issued against. **A refit is covered by the continuous half; a structural change is not.**

This is what the literature means by continual validation: for a twin, validation must be continuous and recalibration triggered when the model stops matching the world, rather than a one-off finished at handover [1]. Chapter 7 Sec. 7.4.7 introduced the idea; this is the schedule.

14.3.4 What changes when the model is learned

Chapter 8 Sec. 8.12 said every retraining trigger in its Sec. 8.6.3 is a Chapter 14 workflow. Four things change and none of them is the pipeline.

The diff at step 4 stops being readable. Three physics parameters diff to three numbers. New weights diff to nothing a human can read, so **the golden trajectories of Sec. 14.2.4 stop being a supplementary check and become the only readable diff you have.** Chapter 8 Sec. 8.2.3 said this; here it becomes the release gate's justification.

The release now has three artifacts, not one. Code, weights and training data, each by content hash (Chapter 8 Sec. 8.2.3), and the training dataset must be a Chapter 10 Sec. 10.7.2 definition rather than a copy – otherwise the release is not reproducible and Chapter 7 Sec. 7.3.4's precondition fails.

Retraining resets the credibility argument entirely, where a refit amends it. Chapter 8 Sec. 8.5.2 made this point; operationally it means a retrain is a *quarterly-cadence* event in Sec. 14.3.3's table, not a continuous one – which caps how often you may retrain, and that cap is a real constraint people discover late.

One trigger runs backwards. Chapter 8 Sec. 8.6.3's third condition – the training data's certification as "normal" is questioned – invalidates decisions already made. Sec. 14.1.2's register needs an `on_fire` for it that is not "retrain": it is Chapter 10 Sec. 10.6.4's five-step traversal, followed by notifying whoever consumed the affected outputs. **Write that runbook before you need it**, because you will be writing it under pressure otherwise.

14.3.5 The announced change, and whether it has an answer

Chapter 10 Sec. 10.9.5 admitted this in the credibility argument itself: *changes to the physical twin enter the record only when a human enters them; an unrecorded repotting is invisible to every check in this document.* Chapter 10 called it the one thing infrastructure cannot fix. Sec. 14.1.3 then found that **about half of the book's twenty-four expiry conditions are in this category.**

So: does it have an operational answer? **Partly, and it is worth being precise about which part.**

What genuinely works. Attach the record to the act, not to the person. If repotting requires a form because the form is how you get a new pot from the store, the record happens. If it requires goodwill after the fact, it does not. **This is a process-design observation and it is the only reliable one available**: the change is recorded when recording it is on the path of least resistance for the person doing the work.

What partly works. Reconciliation. Once a quarter, walk the rig with the context table and check it. Cheap, finds real errors, and finds them late.

What does not work, despite being the usual proposal. Detecting the change from the data. Chapter 7 Sec. 7.7.2 already noted the symptom – a step change in the residuals a fortnight later, by which time the twin has been quietly wrong. And Chapter 9 Sec. 9.7.4 showed the general shape of this failure: the detector needs the model, the model is what the change invalidated, and a monitor that fires *after* the damage is a postmortem rather than a trigger.

> **The honest statement, which belongs in the credibility argument.** Some expiry conditions have no detector. Name them, say who is expected to announce them, and add a reconciliation check. **A register row whose detector is "a human remembers" is not a defect in the register – it is the register doing its job, which is to make that visible.**

14.4 The operating bill

Chapter 1 Sec. 1.4 called model maintenance "the cost most business cases omit entirely" and Chapter 1 Sec. 1.14 said this chapter is where it comes due. Thirteen chapters later, here is the invoice.

14.4.1 Chapter 1's estimate, restated

```
RUN (annual)
  Small server / hosting                                        EUR   600
  Model + threshold maintenance, 0.5 day/month x EUR 400         EUR 2,400
                                                               ---------
                                                                EUR 3,000
```

Six engineer-days a year. That was a reasonable guess by somebody who had not yet designed the system. Now we have designed it.

14.4.2 The real list, counted from the book's own design

Engineering days only. Lab-operations days – walking to the rig, replacing a dripper – are the researcher's time and are counted in Chapter 1's benefit line, not here.

Activity	Frequency	Days each	Days/year	Source
Recalibration and its re-validation	13/year	0.5	**6.5**	Ch7 Sec. 7.7.2's four-week expiry; Sec. 14.3.2's nine steps

Activity	Frequency	Days each	Days/year	Source
Alert handling, engineering share	13/year	0.1	**1.3**	Ch11 Sec. 11.9.3's queue; recording the resolution and reviewing the detector
Cannot-evaluate digest review	Monthly	0.25	**3.0**	Ch11 Sec. 11.4.6's 175/year, aggregated
Sensor recalibration or replacement	2/year	0.5	**1.0**	Ch9 Sec. 9.2.2's drift figure; the context row, the binding, the re-validation
Dependency and platform upgrades	4/year	0.5	**2.0**	Ch12 Sec. 12.5.4's operations line; Ch13's framework maintenance
Incident investigation	2/year	1.0	**2.0**	Ch7 Sec. 7.7.3's five questions
Recording physical changes; quarterly reconciliation	6/year	0.25	**1.5**	Sec. 14.3.5
			17.3	

```
17.3 engineer-days x EUR 400 = EUR 6,920
plus hosting                 = EUR   600
                               -----------
true RUN                     = EUR 7,520 per year
```

Against Chapter 1's EUR 3,000. A factor of 2.5 in money and 2.9 in days.

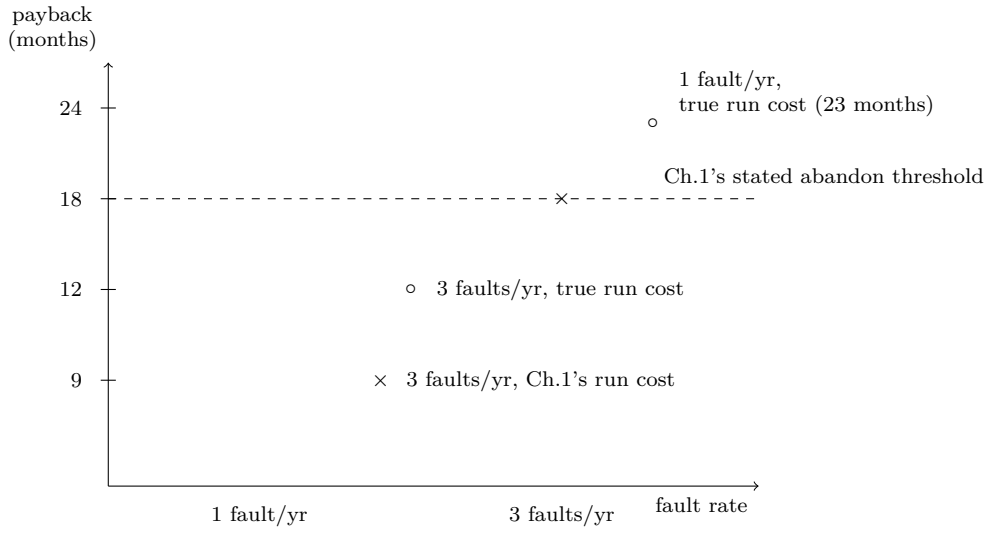
And notice where it comes from. Recalibration alone is 6.5 days – **more than Chapter 1's entire annual allowance** – and it is not an overrun or a surprise. It is the direct consequence of Chapter 7 Sec. 7.7.2 choosing a four-week expiry, a decision made in a chapter that never costed it.

14.4.3 Chapter 1's payback, recomputed

Chapter 1's other figures stand: build EUR 12,000, benefit EUR 19,600 a year.

```
Chapter 1:   (12,000 + 3,000) / 19,600 = 0.77 years = about  9 months
Corrected:   (12,000 + 7,520) / 19,600 = 1.00 years = about 12 months
```

Nine months becomes twelve, and the project survives. That is worth saying plainly: the omitted cost is real, it is large in proportion, and this particular case absorbs it. A sponsor told twelve months at the outset would still have approved it. Figure 14.3 shows why the *falsifier* does not survive as comfortably, which is Sec. 14.4.4's result in advance.



× = computed with run cost EUR 3,000 (Chapter 1)
∘ = computed with run cost EUR 7,520 (the real bill)

the healthy case absorbs the correction (9 → 12).
the marginal case does **NOT**: 18 → 23, well past the threshold Chapter 1 set for itself.

Figure 14.3 A run-cost error is not uniform. It barely moves a strong case and it moves a marginal one across the line the value case drew.

That asymmetry is the practical reason to compute the operating bill early: it changes a *decision* only where the decision was close, which is exactly where it is hardest to notice going wrong.

14.4.4 But Chapter 1's falsifier was computed against the wrong number

This is the sharper result, and it is the one Chapter 1's own method demands.

Chapter 1 Sec. 1.8.8 wrote a falsifier, correctly, and it said:

> The case depends on the fault rate. At 3 faults per year the payback is 9 months. At 1 fault per year the benefit falls to roughly EUR 10,000 and payback stretches to about 18 months, at which point [. . .] the honest recommendation is a cheaper alarm instead of a twin.

Check the arithmetic: `(12,000 + 3,000) / 10,000 = 1.5 years = 18 months`. Correct – **against a run cost of EUR 3,000.**

With the true run cost:

```
(12,000 + 7,520) / 10,000 = 1.95 years = about 23 months
```

So the one-fault case is not marginal, it is well past the point Chapter 1 itself identified as the abandon threshold. **Where is the real boundary?** Chapter 1's benefit scales with

the fault rate `f` at roughly EUR 4,800 per fault above a fixed EUR 5,200 of inspection saving, which reproduces both of Chapter 1's figures:

```
benefit(f) = 4,800 f + 5,200      benefit(3) = 19,600   benefit(1) = 10,000
```

Setting payback to Chapter 1's own 18-month abandon threshold:

```
(12,000 + 7,520) / benefit(f) = 1.5
                   benefit(f) = 13,013
                       4,800f = 7,813
                            f = 1.6 faults per year
```

> **Chapter 1 put the abandon point at one fault a year. It is really about 1.6.** The project is more fragile than the chapter that proposed it believed, and the reason is entirely the run cost – not the build, not the model, not anything Part II or Part III got wrong.

Two things follow, and both are about method rather than about pots.

A falsifier computed against an incomplete cost is itself incomplete. Chapter 1 Sec. 1.8.8 was right to demand one and right about its structure. This is what it looks like when the demand is honoured thirteen chapters later and the answer moves.

This is why the cheapest test in Chapter 1 is still the right one. Chapter 1 said: before writing any code, count how many doses produced no moisture step. At 3 faults a year the case is comfortable; at 1.6 it is not. **That afternoon of querying is worth more now than it was when Chapter 1 proposed it**, because the margin it is testing is narrower than Chapter 1 thought.

14.4.5 What to automate first, decided by the bill

Not by preference. Read Sec. 14.4.2's right-hand column and automate the largest line.

Recalibration, at 6.5 days a year, is 38 per cent of the bill, and Sec. 14.3.2 found eight of its nine steps mechanisable. Automate steps 1 to 3 and 5 to 9, leave step 4 – the human reading three numbers – and the activity drops to about 0.15 days each:

```
13 x 0.15 = about 2 days/year, from 6.5.  Saving: 4.5 days = EUR 1,800/year.
Build cost: about 5 engineer-days = EUR 2,000.
Repays in about 13 months, and every year after is EUR 1,800 cheaper.
```

And it moves the number that matters:

```
new RUN = (17.3 - 4.5) x 400 + 600 = EUR 5,720
new abandon threshold: (12,000 + 5,720) / 1.5 = 11,813 = benefit(1.38)
```

The abandon threshold moves from 1.6 faults a year back to about 1.4. Modest, real, and computed rather than asserted – which is the whole point of doing the arithmetic before choosing what to build.

What not to automate first. The cannot-evaluate digest, at 3 days a year, looks like the second-largest line and is the wrong target: it is already aggregated, and what it costs is *attention*, which automation does not return. **Reducing it means fixing the connector so there is less to review**, which is a Chapter 9 activity that Sec. 14.4.2 has just given a price.

14.5 Evolution: when the twin's purpose changes

14.5.1 Parts change, and purposes change, and they fail differently

Everything so far has been about parts: models refitted, sensors replaced, platforms upgraded, standards versioned. There is a second kind of change and the book's machinery has not yet been pointed at it.

The evolution literature makes the distinction explicit. Twin evolution spans changes in the actual twin – both discontinuous, such as component replacement after a failure, and continuous, such as wear [2] – and, separately, the twin's own lifecycle as a software system with its own evolution, which the same work argues should be treated as a core principle rather than an afterthought [2]. The purpose dimension is sharper still: twins are developed for a specific purpose, and the constant evolution of systems, users' needs and environments demands that the twin adapt – a family home's twin whose purpose is "expanded, changed and re-prioritized throughout its ongoing lifecycle" is the worked case [3].

Why the second kind is harder. A part change has a trigger. The pot was repotted; somebody can announce it. **A purpose change frequently has no event at all** – it happens because a user starts asking a different question of the same screen.

14.5.2 The credibility argument that is true and irrelevant

Here is the failure this section exists for.

Chapter 7's credibility argument states a claim, an intended use, evidence, limits and an expiry. Now suppose the *intended use* silently changes: the researchers, who were using the twin to catch failed doses, start using its moisture plot to decide when to harvest.

Nothing has expired. The parameters are current, the monitors are quiet, the hold-out spread is still 2.0, the argument is **entirely true**. And it is about a decision nobody is making any more.

> **A credibility argument that is true and irrelevant is worse than one that has expired, because nothing fires.** An expired argument has a trigger, an owner and a workflow. A true-and-irrelevant one has a happy dashboard.

Why this is not a hypothetical. Chapter 7 Sec. 7.1.2 established that how much evidence you owe depends on what the twin is allowed to do, and Chapter 1's whole method derives fidelity from the value metric. **Both of those are anchored to a purpose**, so a purpose that moves invalidates the evidence requirement without touching the evidence.

The detector, and it is a question rather than a monitor. Add one row to Sec. 14.1.2's register whose condition is *the twin is being used for a decision not named in the credibility argument's intended-use statement*, whose detector is the quarterly review, and whose check is one question to one user: **what did you last use this for?** It takes five minutes and it is the only instrument available.

14.5.3 Four purpose changes, and what each breaks

Change	What still works	What silently breaks
The same output feeds a **new decision**	Everything technical	Chapter 7's accuracy target was derived from the old decision (Sec. 7.5.2). The 16 per cent detection floor may be unacceptable for the new one
Advisory becomes acting	Everything technical	Chapter 2 Sec. 2.8's obligations, Chapter 3's actuation guard, and Chapter 7's evidence bar all step up. Chapter 11 Sec. 11.6.4's four rules become mandatory
A second asset is added	The code	Chapter 12 Sec. 12.8.4's expiry fires; the build-versus-buy arithmetic changes; Chapter 13's counterparty count may become non-zero
A new audience arrives (an auditor, a regulator, a customer)	Everything technical	Chapter 13 Sec. 13.2.2's *given* case begins, and the evidence you have must be re-presented in somebody else's shape

Notice the column headings. In every row the technology is fine. That is what makes purpose drift the hardest change to notice and the easiest to under-react to, and it is why it belongs in an operations chapter rather than a design one.

14.5.4 Retiring a twin

Rarely discussed, and it is a real activity with real obligations.

Three reasons a twin ends. The physical twin is decommissioned. The decision it served stopped being made. Or the value case stopped holding – Chapter 1 Sec. 1.7's stop conditions, arriving late instead of early, and Sec. 14.4.4's corrected threshold makes that outcome more likely than Chapter 1 suggested.

What retirement obliges you to do, and only the first is obvious:

Stop it acting. If there is a command path, disable it first and separately, before anything else, because a half-retired twin still holding an actuation credential is the worst configuration available.

Keep the record. Chapter 10's provenance and Chapter 7's arguments may be needed after the twin is gone – for an incident review, an audit, or the next twin. **Retention outlives**

the system, and a retirement plan that deletes the store has destroyed the evidence for decisions already taken.

Write down what was learned. The calibration values, the assumption ledger that survived contact with reality, the incidents. Sec. 14.6.1's practices and Chapter 13 Sec. 13.5.3's exemplars are both arguments that this is the cheapest thing a twin project produces and the most reusable.

14.6 Practices worth naming

14.6.1 TwinOps, and DevOps for twins

The practice has a name and a small literature, and both are worth recognising rather than adopting wholesale.

TwinOps couples model-based engineering, DevOps and digital twins: generating and deploying a system, its simulation and its twin through one pipeline, and collecting runtime execution traces to compare against the engineering artifacts – model simulation and analysis – so that diagnosis is rapid [4], [5]. **The idea worth taking is that comparison**, and this book has been building it since Chapter 1: the residual is a runtime trace compared against a model, and Sec. 14.2.3's simulate-the-physical-twin is the same comparison run the other way.

DevOps for twin development and evolution is being worked on directly, with an emphasis on the iterative development and evolution of twins rather than their initial construction [6], [7], and the evolution literature explicitly asks how DevOps practice can support twin evolution and calls for evolution to be treated as a core principle [2].

What this chapter takes and what it leaves. Takes: the framing that a twin is a software system with its own lifecycle, and that the interesting part is the second year rather than the first. Leaves: pipeline mechanics. Chapter 12 refused to name container technology and the same rule applies – your existing delivery practice works, and what is different is the four artifacts of Sec. 14.2.1 and the absent staging environment of Sec. 14.2.2.

One empirical note worth carrying into a planning meeting. An interview-based study of continuous integration and delivery for cyber-physical systems found the hardware-in-the-loop stages to be the awkward part [8]. Chapter 7 Sec. 7.3.3 already flagged this as permission to expect the physical stages to be harder than a web service's, and it is the single most reliable source of schedule surprise in this chapter's subject.

14.6.2 Replay

Chapter 5 Sec. 5.12 sent this here. **Replay is re-running the twin over recorded inputs to reproduce a past state or decision.**

It is the operational payoff of three earlier chapters' discipline. Chapter 5 Sec. 5.6's recorded-inputs requirement, Chapter 10's retained derived output with its `decision_time` and `inputs_watermark`, and Chapter 11 Sec. 11.8.2's rule that re-evaluation produces a new row rather than an edit. With all three, replay is a query and

a run. Without any one of them it is impossible.

Three uses, in increasing order of value.

Incident review. Chapter 7 Sec. 7.7.3's five questions, answered by re-running rather than by reasoning.

Regression testing after a model change. This is golden trajectories (Sec. 14.2.4) with real history instead of recorded scenarios – more realistic, and it grows for free.

Answering "what would the new version have done?" Replay the last quarter through the candidate model and count the alerts it would have raised. **For the demonstrator that is Chapter 11 Sec. 11.4.5's alert economics computed on real data instead of assumptions**, which is the strongest release evidence available and costs one afternoon.

14.6.3 Fault injection

Chapter 11 Sec. 11.7.2 named it and sent it here, correctly, because the literature frames it as a validation technique for dependability rather than as a twin service [9].

Operationally it is Sec. 14.2.3 with intent. Generate the failures you cannot wait for – a dose that delivers nothing, a stuck sensor, a gateway that stops, a clock that jumps – and confirm the system does what the credibility argument says it does.

Two rules.

Inject into the simulated physical twin, never the real one, unless the physical twin is genuinely expendable and somebody senior has agreed.

Derive the injection list from the credibility argument's limitations section. Chapter 7's limits paragraph names the things the twin claims not to handle; **fault injection is how you find out whether it handles them the way you claimed it would fail.** That is a much better source of test cases than imagination.

14.6.4 What security looks like in operation

Chapter 3 Sec. 3.3.3 owns security and said Chapter 14 returns to running it in production. One subsection, and the rest stays where it was.

Three things change once the twin is running, and all three are consequences of chapters you have already read.

The credential inventory grows quietly. Every connector, every service, every platform integration. Chapter 12's platform decision and Chapter 13's open-source dependencies each added some. An expiry register row for credential rotation costs nothing and is the kind of thing that is missing in a postmortem.

Provenance is a security property, not only an engineering one. Chapter 10 Sec. 10.6.4 made this point via the layered threat classification, in which integrity loss in the acquisition layer propagates into synchronisation and simulation quality [10]. Operationally: **the record that lets you answer Chapter 7's five questions is the same record that lets you answer "what did the attacker change?"**

The command path is the asset. Chapter 3's guard, Chapter 11 Sec. 11.6.4's four rules, and the retirement rule of Sec. 14.5.4 all point at one thing. If you review one security control a year, review that one.

14.7 Worked example: a year of operating the demonstrator

Chapter 1 costed a year of running this twin in two lines. Here is what the year actually contains.

14.7.1 The register, assembled

Twelve rows for this twin, drawn from Sec. 14.1.1 and classified per Sec. 14.1.3:

#	Condition	Kind	Detector	Owner
1	4 weeks since calibration	Scheduled	Calendar job	Twin owner
2	Residual bias outside plus or minus 1.0 over 14 days	Self-firing	Ch7 monitor	Twin owner
3	Residual spread growing	Self-firing	Ch7 monitor	Twin owner
4	Estimator gain outside 0.4 to 0.85 for 48 h	Self-firing	Ch7 monitor	Twin owner
5	Cannot-evaluate rate rising	Self-firing	Ch11 digest trend	Connector owner
6	Evaluation queue backing up	Self-firing	Ch11 Sec. 11.8.3	Twin owner
7	Pot repotted, plant replaced, dripper moved	**Announced**	The lab's pot-request form	Lab operations
8	Sensor replaced or recalibrated	**Announced**	The lab's parts form	Lab operations
9	Sensor drift margin consumed	Scheduled	Calendar, from Ch9 Sec. 9.2.2	Twin owner
10	Model or parameter version changed	Self-firing	Release pipeline	Twin owner
11	A second bench is funded	**Announced**	Quarterly review	Twin owner

#	Condition	Kind	Detector	Owner
12	The twin is used for a decision not in the intended-use statement	**Announced**	Quarterly review: one question to one user	Twin owner

Four announced rows out of twelve, and rows 7 and 8 are attached to forms people already fill in to get the parts – which is Sec. 14.3.5's only reliable mechanism, applied.

14.7.2 The year, month by month

	Activity
Every 4 weeks	Recalibration and re-validation, Sec. 14.3.2's nine steps (13 times)
Monthly	Register review, 15 minutes. Cannot-evaluate digest trend
Weekly	Glance at the three monitors
As they happen	About 13 alerts; 6 recorded physical changes
Quarterly	Full re-validation; context reconciliation walk; the purpose question
Twice	A sensor event; an incident
Four times	A dependency upgrade

14.7.3 The bill, and what the sponsor is told

From Sec. 14.4: **17.3 engineer-days, EUR 7,520 a year**, against the EUR 3,000 in the original case. Payback 12 months rather than 9.

What to say, and when. Say it at the outset, in the value case, not in year two. Chapter 1's canvas has a RUN line and it should carry this number from the start, because a sponsor told twelve months up front approves the project and a sponsor told nine and charged twelve does not fund the next one.

And say the falsifier honestly. Below about 1.6 faults a year this twin is not worth running (Sec. 14.4.4), which is a narrower margin than Chapter 1 stated. **The afternoon of querying that Chapter 1 recommended is therefore still the right first action**, and this is the second time the book has arrived at that conclusion from a different direction.

14.7.4 What the year changes about the credibility argument

Chapter 7 Sec. 7.8.5's argument gains one line and loses none:

Operations. Parameters are refitted every four weeks by an automated workflow with a human diff review; the hold-out is re-run at each refit and the full validation quarterly. Twelve expiry conditions are held in a register reviewed monthly, of which **four have no automatic detector and depend on a human announcement**; two of those four are attached to forms the lab already completes. A quarterly context reconciliation checks the register against the rig.

That paragraph is what an auditor would actually want, and no chapter before this one could have written it.

14.8 Faded example: the turbine fleet in year three

Chapters 4 through 13 each took the turbine one step further. Now it has been running for three years. Two parts are worked; four are yours.

The system, recapped. Eighty floating turbines, a platform-based build, four services, a certification output facing a regulator, and Chapter 13's non-zero counterparty count. Diagnostic twins of this shape run on operational assets [11] and are validated against full-scale prototypes [12].

(a) The announced-trigger problem is worse and has a better answer – worked. Sec. 14.1.3 found about half the conditions announced, and Sec. 14.3.5's reliable mechanism was to attach the record to the act. Offshore, the acts are *maintenance work orders*, and those already exist in a system, with dates, assets and parts. **So the mechanism is available and industrial-strength: subscribe to the work-order feed and turn relevant orders into Chapter 10 context events.** A mooring line replaced is a work order before it is anything else.

The general lesson, which transfers to the pot. The announced category is not hopeless – it is a question of whether the organisation already records the act somewhere for its own reasons. Look for that record before designing a new form.

(b) The re-validation cadence is the binding constraint – worked. Sec. 14.3.3 split validation into continuous and full. For eighty assets the full half cannot be quarterly per asset without a dedicated team: eighty assets at even one day each is eighty days a quarter. So the fleet needs something the demonstrator did not: **a validation policy that is per-population rather than per-asset**, with full re-validation on a sample plus continuous scoring on all – and an explicit rule for when an individual asset is promoted out of the sample into full treatment. Industrialising twin production puts new demands specifically on the speed of validation [13], and this is where that pressure lands.

Now yours.

(c) Run Sec. 14.4.2's bill for one turbine twin and then for the fleet. State which lines scale with the number of assets and which do not, and use that split to say where a platform's operations offering (Chapter 12 Sec. 12.5.4) earns its money.

(d) Sec. 14.2.2 said the physical twin has no staging environment. For a turbine, name what stands in and what it cannot test. *Hint: Chapter 7 Sec. 7.9(b) established*

that validating a reduced-order model against the full finite-element model validates the reduction, not the physics – the same distinction applies here.

(e) Sec. 14.5.2's true-and-irrelevant failure needs a detector that is a question to a user. Who do you ask, on a fleet of eighty, and how often? Then say what happens to the certification output if the answer reveals that the maintenance schedulers stopped using it eighteen months ago.

(f) Sec. 14.5.4's retirement obligations, for one turbine decommissioned out of eighty while the fleet twin continues. Say what must be kept, for how long, and which of Chapter 10's tables cannot have that asset's rows deleted at all.

14.9 Posed problem: the handover

No solution is given. This is the deliverable, and it is the one you are most likely to be asked for at a moment when you have no time to think.

The situation. You built the district-heating twin of Chapters 9 to 13 and you are leaving the project in six weeks. It will be operated by a team of two who did not build it: one experienced developer with no twin background, and one recent graduate. There is no on-call rotation. The economic regulator's next review is in fourteen months.

Produce a handover package of no more than six pages containing:

1. **The expiry register** (Sec. 14.1.2), populated. Every condition from the design documents of Chapters 9 to 13, classified by Sec. 14.1.3's three kinds, with a detector and an owning role for each. Say explicitly how many rows have no automatic detector.
2. **The operating loop** (Sec. 14.3.1): the five activities, with cadences and time estimates, in a form the new team can put in a calendar.
3. **The operating bill** (Sec. 14.4.2) for this system, with the engineering days separated from operations days, and an honest statement of whether two people can absorb it alongside their other work.
4. **The release process** (Sec. 14.2): what the four artifacts are here, what stands in for staging given three meter vendors, and what the gate is.
5. **Three runbooks**, one page each: a recalibration, a retroactive-invalidation traversal (Chapter 10 Sec. 10.6.4, per Sec. 14.3.4's last point), and the regulator's evidence request.
6. **The purpose-drift question** (Sec. 14.5.2): who to ask, how often, and what the current intended-use statement actually says.
7. **What you would switch off before leaving**, and why. *There is almost certainly something – a service nobody uses, a monitor nobody reads, a report nobody opens – and leaving it running transfers a maintenance cost with no benefit.*
8. **The one thing that will break first**, named, with what to do about it.

What a good answer looks like. It counts the announced triggers and says the number out loud, because that number is the handover's biggest risk. It sizes the bill against two people and is honest if the answer is that they cannot absorb it – **a handover that pretends the workload fits is worse than one that says it does not**, because the second gets a conversation and the first gets a failure in month four. It attaches the

announced triggers to records the utility already keeps, per Sec. 14.8(a), rather than inventing forms. It switches something off. And its final answer – the thing that will break first – is specific and probably unglamorous: most likely the recalibration job, silently, because Sec. 14.1.3 says scheduled triggers fail by quiet omission and nobody notices a job that stopped running.

14.10 Summary

Seven things, tied to the five objectives.

1. **This book wrote twenty-four expiry conditions across seven chapters and built nothing to watch them** (Sec. 14.1.1). Chapter 7 said expiry conditions are only worth writing if something acts on them, and then did not act. The register is one list, one owner, one monthly review, and its value is entirely in the aggregation. *(Objective 1.)*
2. **Only one of the three kinds of trigger fires by itself** (Sec. 14.1.3). Roughly nine are self-firing, three scheduled, and **about half are announced** – they exist only if a human says something, and the boundary is worth arguing about, because **an announced trigger you can promote to self-firing stops depending on memory forever.** The reliable mechanism is to attach the record to the act, so that recording is on the path of least resistance; reconciliation is the partial fallback; and detecting the change from data does not work, because the monitor fires after the damage. *(Objective 1.)*
3. **You ship four artifacts and the physical twin has no staging environment** (Sec. 14.2). Three of the four change without anybody writing code. Simulate-the-physical-twin tests the plumbing with ground truth in seconds and **cannot test the physics, because it shares the model's errors**; golden trajectories are the gate; shadow deployment is cheap because Chapter 10's schema already supports two beliefs about one moment; and what cannot be rolled back is what the twin already did. *(Objective 2.)*
4. **The cadence problem dissolves once validation stops being one activity** (Sec. 14.3.3). Continuous scoring runs at the speed of the data and covers a refit; full re-validation runs quarterly and covers a structural change. A retrain is a quarterly-cadence event, which caps how often you may retrain. *(Objective 2.)*
5. **The operating bill is 17.3 engineer-days, not six** (Sec. 14.4). Chapter 1 guessed EUR 3,000 a year; the design it did not yet have costs EUR 7,520. Recalibration alone – 6.5 days – exceeds Chapter 1's entire allowance, and it is the direct consequence of Chapter 7 choosing a four-week expiry in a chapter that never costed it. Payback moves from nine months to twelve, and the project survives. *(Objective 3.)*
6. **Chapter 1's falsifier was computed against the wrong number** (Sec. 14.4.4). Chapter 1 put the abandon threshold at one fault a year; with the true run cost it is about 1.6. **The project is more fragile than the chapter that proposed it believed**, and the cheapest test in the book – an afternoon counting failed doses – is worth more now than when Chapter 1 recommended it. *(Objective 3, 5.)*
7. **Parts change and purposes change, and only one of them has a trigger** (Sec. 14.5). A credibility argument that is *true and irrelevant* is worse than an

expired one, because nothing fires: the parameters are current, the monitors are quiet, and the claim is about a decision nobody is making. Its only detector is a question to a user, quarterly, and it takes five minutes. *(Objective 4.)*

And the note this chapter adds to Part III. Chapters 11, 12 and 13 each reduced a technology question to a question about the situation. Chapter 14 does something narrower and, for a book that has spent thirteen chapters recommending things, more uncomfortable: **it prices its own advice.** The four-week calibration cadence, the three monitors, the six quality states, the reproducible datasets and the expiry conditions were all correct recommendations, and together they cost three times what Chapter 1 budgeted for running the system. **A method that never audits its own cost is a method that has not been used**, and Sec. 14.4 is this book's audit of itself.

14.11 Exercises

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

14.11.1 *(Objective 1.)* Classify each trigger as self-firing, scheduled or announced, and name a detector: (a) the model's residual bias exceeds a threshold; (b) a supplier changes the units its sensor reports; (c) six months have passed since the last full validation; (d) the twin's only modelling expert leaves; (e) the input-coverage refusal rate doubles.

14.11.2 *(Objective 1.)* Your register has 18 rows, of which 11 are announced. Give three things you would do, in order, and say which of the 11 you would attack first and why.

14.11.3 *(Objective 2.)* You are releasing a refitted parameter set. List the gates it must pass, in order, and say which one would catch each of these: (a) the fit was run on a month with a two-week outage; (b) the new `k_day` is 6.1 instead of 4.28; (c) the change is fine but the deployment script points at the wrong binding.

14.11.4 *(Objective 2.)* Explain, in three sentences, why a simulate-the-physical-twin test suite that passes completely tells you nothing about whether the model is right – and name the one thing it does tell you that Chapter 7's hold-out does not.

14.11.5 *(Objective 3.)* Recompute Sec. 14.4.2's bill for a twin whose calibration expiry is 12 weeks rather than 4, with everything else unchanged. Then compute the new payback and abandon threshold, and say what you would need to establish before recommending the longer cadence.

14.11.6 *(Objective 3.)* Your operating bill is 40 engineer-days a year, of which incident investigation is 12, recalibration is 9, and digest review is 8. Your automation budget is 10 engineer-days. Say what you would build and justify it from the bill – then say what additional information would change your answer.

14.11.7 *(Objective 4.)* For each, say whether it is a part change or a purpose change, and which of Sec. 14.5.3's rows applies: (a) the greenhouse adds twelve more pots; (b) the twin's moisture plot starts being used to schedule harvesting; (c) the moisture sensor is replaced with a different model; (d) the lab is bought by a company that wants the twin to control watering automatically.

14.11.8 *(Objective 4.)* Write the intended-use statement check as a question you would

actually ask a user, plus the two follow-ups you would ask if the answer is not what the credibility argument says. Then say what you would do if a user answers with a decision the twin's accuracy cannot support.

14.11.9 *(Objective 5.)* Sec. 14.4.5 automated recalibration because it was the largest line. Give the case *against* automating it first, using Sec. 14.3.2's step 4 and Sec. 14.1.3's failure mode for scheduled triggers.

14.11.10 *(Objectives 1-5, and the one that leaves the book.)* Take whatever you built in Chapters 9 to 13's exercises. Write its expiry register: every condition, classified, with a detector and an owner. Count the announced rows. Then estimate its operating bill in engineer-days and compare that with what you would have guessed before doing the exercise.

Solutions and hints

14.11.1. (a) Self-firing; Chapter 7 Sec. 7.7.4's rolling bias monitor. (b) **Announced, and this is the trap** – Chapter 9 Sec. 9.8 put `unit` in the tuple, so a *daily job comparing each series' unit against the previous day* turns it into a self-firing trigger for about an hour of work. **Some announced triggers can be promoted, and finding those is the highest-value thing you do with the register.** (c) Scheduled; a calendar job, and note Sec. 14.1.3's failure mode – something must check that the job ran. (d) Announced, and there is no detector; the mitigation is Sec. 14.1.2's rule that owners are roles. (e) Self-firing; Chapter 8 Sec. 8.6.2's coverage check.

14.11.2. In order: (i) **try to promote some** – ask of each whether any data the twin already holds would reveal it, as in 14.11.1(b); (ii) **attach the rest to acts** rather than to memory, per Sec. 14.3.5, looking for records the organisation already keeps for its own reasons; (iii) add a reconciliation check for whatever remains and **state the residue in the credibility argument**. Which first: the one whose consequence is worst and whose silence lasts longest – usually a physical change that invalidates the model, because Chapter 7 Sec. 7.7.2's expiry fires on it and the twin is confidently wrong until somebody notices.

14.11.3. Gates in order: dataset materialisation and the degraded-window refusal (Sec. 14.3.2 step 2); the human diff at step 4; the hold-out at step 5; golden trajectories at step 6; then deployment. (a) Caught by the degraded-window refusal – **and it is a Chapter 9 finding, not an error**. (b) Caught by the human diff: 6.1 against 4.28 is a 43 per cent move in a parameter that should be stable, and no automated threshold would have known that; this is why step 4 stays human. (c) Caught by **none of them** – which is the point of the exercise. A binding error is Chapter 3 Sec. 3.3.1's highest-consequence, lowest-visibility failure, no test fails, and the only defences are the daily orphan-check of Chapter 10 Sec. 10.2.4 and reading the deployment diff.

14.11.4. The synthetic measurements were generated by the same model the twin uses, so any error in that model is present identically on both sides of the comparison and cancels; a passing suite therefore establishes that the pipeline transports and processes numbers correctly, not that the numbers describe the world. What it tells you that the hold-out does not: **how the system behaves under failures that have not happened yet** – a stuck sensor, a missing dose, a clock jump – because those can be generated on demand and cannot be waited for.

14.11.5. At 12 weeks the recalibration count drops from 13 to about 4.3 a year, so 6.5 days becomes about 2.2 – a saving of 4.3 days. New total about 13 days, `13 x 400 + 600 = EUR 5,800`. Payback `(12,000 + 5,800) / 19,600 = 0.91 years`, about 11 months. New abandon threshold: `(12,000 + 5,800) / 1.5 = 11,867 = benefit(1.39)`, so about 1.4 faults a year. **What you must establish first:** that the parameters actually hold for 12 weeks. Chapter 4's assumption A4 claimed weeks and Chapter 7 Sec. 7.5.3 admitted only three weeks were tested with four claimed. Extending to twelve is an extrapolation of a factor of four – **so the prerequisite is evidence, and the cheapest form of it is to keep refitting every four weeks for a year and look at how much the parameters actually moved.** That costs nothing extra, because you were doing it anyway.

14.11.6. Incident investigation is the largest line at 12 days but is the least automatable, because each incident is different – what reduces it is *fewer incidents* or *better provenance*, and Chapter 10 already built the second. Recalibration at 9 days is the same case as Sec. 14.4.5 and is the obvious build. Digest review at 8 days is attention, not work, and automation does not return attention. **So: automate recalibration (about 5 days of the budget), and spend the remaining 5 on whatever Chapter 10 provenance is missing, because that reduces the 12-day line.** What would change the answer: **the distribution of the 12 days.** If two incidents took six days each, that is a systemic problem worth fixing directly; if twelve took one day each, the provenance investment is right.

14.11.7. (a) Part change, though at some point it becomes Sec. 14.5.3's third row and fires Chapter 12 Sec. 12.8.4's expiry. (b) **Purpose change**, first row – and the accuracy target derived in Chapter 7 Sec. 7.5.2 was for fault detection, so it may be wholly unsuitable for harvest scheduling while remaining entirely accurate. (c) Part change, Chapter 9 Sec. 9.2's territory, unless the response curve differs, in which case it is also a model change. (d) **Purpose change**, second row, and the most expensive one in the book: advisory becomes acting, so Chapter 2 Sec. 2.8, Chapter 3's guard, Chapter 7's evidence bar and Chapter 11 Sec. 11.6.4 all step up at once.

14.11.8. The question: *"What was the last decision you made using this?"* – open, past-tense and specific, rather than "what do you use it for", which invites a description of the intended use they were told about. The two follow-ups: *"What would you have done without it?"* and *"How accurate do you need it to be for that?"* If the answer names a decision the accuracy cannot support: **do not remove access, and do not settle for a warning.** Take it back to Chapter 1's method – name the decision, derive the fidelity requirement, and either meet it or state plainly, in the credibility argument and to that user, that the twin does not support it. A warning nobody reads is the outcome to avoid.

14.11.9. The case against: step 4 is the human diff, and an automated pipeline that runs every four weeks makes it very tempting to automate step 4 too – at which point a parameter drifting steadily is accepted thirteen times a year and nobody sees the trend that Chapter 7 Sec. 7.4.7 wanted the parameter history for. And Sec. 14.1.3 says **scheduled triggers fail by quiet omission**: a manual recalibration that stops happening is noticed within a month, whereas an automated job that stops is noticed when something else breaks. **The mitigation, if you automate anyway:** make the job's *last successful run* an expiry register row of its own, and put the step-4 diff in front of a human as a required approval rather than a notification.

14.11.10. No solution. One prediction: your bill will be larger than your guess, and the largest single line will be something you decided in an earlier chapter without costing it – which is exactly what happened to Chapter 7's four-week cadence, and the reason this chapter exists.

14.12 Where to go next

In this book. One chapter remains. **Chapter 15 takes every number in this chapter and multiplies it by `N`** – and the interesting part is that they do not all scale the same way. Sec. 14.4.2's bill has lines that are per-asset (recalibration, alert handling) and lines that are per-system (dependency upgrades, register review), and Chapter 15 is where that split decides an architecture. Sec. 14.1.3's announced triggers scale worst of all, because each asset has its own physical changes and no amount of tooling makes somebody remember. Sec. 14.5's purpose drift scales worst in a different way: with one twin you can ask the user, and with four hundred you cannot. Chapter 12 Sec. 12.5.2's `N` and Chapter 13 Sec. 13.1.3's `C` both become the subject there.

In the literature, if you want more.

- *Evolution as a first-class concern:* [2] is the one to read – it classifies the dimensions along which a twin evolves, separates the actual twin's changes from the twin's own software lifecycle, and explicitly asks how DevOps practice supports it. This book has cited it since Chapter 1 for the continuous/discontinuous distinction; its wider argument is this chapter's subject. [3] is the sharper complement, because it is about **purpose** evolution rather than part evolution, worked on a home twin whose purpose is expanded and re-prioritised over its life.
- *TwinOps and DevOps for twins:* [4] and [5], which put model-based engineering and ordinary delivery practice into a single pipeline and then check what the running system actually did against what the engineering models said it would; [6] and [7] on a DevOps approach aimed specifically at the iterative development and evolution of twins rather than their construction.
- *Why the hardware stages are the hard ones:* [8], an interview study of CI/CD practice for cyber-physical systems. Read it before promising a schedule.
- *Continual validation:* [1], which supplies Sec. 14.3.3's framing and which Chapter 7 Sec. 7.4.7 already leaned on.
- *Fault injection and advanced services:* [9], framed as a dependability technique, which is why it lands in this chapter rather than Chapter 11.
- *Consulted, not drawn on above:* [14] on automated twin testing and detecting performance shift online, [13] on industrialising twin production, [15] and [16] on evolving twins by transfer learning, [17] on model-based DevOps for cyber-physical systems, and [10] for the layered view behind Sec. 14.6.4.

In the demonstrator. Exercise 14.11.10 is the assignment and it is the one that will change how you plan the next project: write the expiry register for whatever you have built, count the announced rows, and estimate the bill. **Then compare the estimate with what you would have said before Chapter 14.** The gap between those two numbers is what this chapter is for, and it is the same gap Chapter 1 identified in its fourth section and could not yet fill.

14.13 References

[1] Z. Ali, R. Biglari, J. Denil, J. Mertens, M. Poursoltan, and M. K. Traoré, "From modeling and simulation to Digital Twin: Evolution or revolution?" *SIMULATION*, vol. 100, no. 7, pp. 751–769, Jul. 2024, doi: 10.1177/00375497241234680.

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

[3] J. Mertens, S. Klikovits, F. Bordeleau, J. Denil, and Ø. Haugen, "Continuous Evolution of Digital Twins using the DarTwin Notation." arXiv, Oct. 2024. Accessed: Nov. 11, 2024. [Online]. Available: http://arxiv.org/abs/2410.23389

[4] J. Hugues, J. Yankel, J. Hudak, and A. Hristozov, "Twinops: Digital twins meets devops," *CARNEGIE-MELLON UNIV PITTSBURGH PA, Tech. Rep.*, 2022, Accessed: Jan. 08, 2025. [Online]. Available: https://apps.dtic.mil/sti/trecms/pdf/AD1168471.pdf

[5] J. Hugues, A. Hristosov, J. J. Hudak, and J. Yankel, "TwinOps - DevOps meets model-based engineering and digital twins for the engineering of CPS," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, in MODELS '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 1–5. doi: 10.1145/3417990.3421446.

[6] F. Bordeleau, A. Motamedi, and É. Poirier, "A DevOps Approach for the Systematic Development and Evolution of Built Assets Digital Twins," Accessed: Nov. 05, 2024. [Online]. Available: https://itc.scix.net/pdfs/w78-2024-paper_114.pdf

[7] S. Aissat, J. Beaulieu, É. Poirier, A. Motamedi, J. Gascon-Samson, and F. Bordeleau, "A devops framework for the systematic engineering and evolution of digital twins for built assets," *Software and Systems Modeling*, Oct. 2025, doi: 10.1007/s10270-025-01330-0.

[8] F. Zampetti, D. Tamburri, S. Panichella, A. Panichella, G. Canfora, and M. Di Penta, "Continuous Integration and Delivery Practices for Cyber-Physical Systems: An Interview-Based Study," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 3, pp. 73:1–73:44, Apr. 2023, doi: 10.1145/3571854.

[9] M. Frasheri, T. Böttjer, P. G. Larsen, L. Esterle, and C. Gomes, "Advanced Digital Twin Services," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds., Cham: Springer International Publishing, 2024, pp. 209–222. doi: 10.1007/978-3-031-66719-0_10.

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

[11] 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.

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

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

[14] Y. Ma *et al.*, "Automated and Systematic Digital Twins Testing for Industrial Processes." arXiv, Feb. 2023. doi: 10.48550/arXiv.2302.13198.

[15] Q. Xu, T. Yue, S. Ali, and M. Arratibel, "Pretrain, Prompt, and Transfer: Evolving Digital Twins for Time-to-Event Analysis in Cyber-physical Systems." arXiv, Apr. 2024. doi: 10.48550/arXiv.2310.00032.

[16] C. Lu, Q. Xu, T. Yue, S. Ali, T. Schwitalla, and J. F. Nygård, "EvoCLINICAL: Evolving Cyber-Cyber Digital Twin with Active Transfer Learning for Automated Cancer Registry System," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Nov. 2023, pp. 1973–1984. doi: 10.1145/3611643.3613897.

[17] B. Combemale *et al.*, "Model-Based DevOps: Foundations and Challenges," in *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, Oct. 2023, pp. 429–433. doi: 10.1109/MODELS-C59198.2023.00076.