
DELETING CODE WITHOUT LOSING KNOWLEDGE

REGENERATIVE SOFTWARE FOR PYTHON LIBRARIES

WORKING PAPER

Thomas Bartz-Beielstein 
bartzbeielstein@gmail.com

2026-10-12

ABSTRACT

In the age of “Software 3.0”, when a complete program can be rewritten in minutes by an agent, the traditional approach of relying on code as the sole source of truth becomes questionable. Fowler advocated for a set of seven primitives, which lay the framework for *Regenerative Software*, so that programs can be deleted and rebuilt without losing their knowledge. The knowledge lives in machine-readable records in a tracked directory, from which a *regeneration bundle* is built for each unit. The regeneration bundle is a single file with the module’s architecture, intent, decisions, public contract, evaluations, and consumers, but no implementation, plus the one dependency module it may import. The open-source package *spotregen* implements these primitives and provides a graphical user interface. This report presents a first application of Fowler’s primitives to the safety-critical forecasting library *spotforecast2-safe*. During the first experiments, the Claude Code session that orchestrated the study consumed more than 43 million tokens, almost all of them its own history re-sent with every request. This motivated the switch to an architecture that is built on local models. Deterministic checks and the maintainer judged the results. The report contains executable examples for every primitive. Each example runs *spotregen* against the repository or against a throw-away copy of it.

Keywords regenerative software • safety-critical software • specification • test oracle • mutation analysis • provenance • time series forecasting

1 Introduction

During the development and application of a safety-critical library many lessons are learned. For example, during the application of the open-source time-series forecasting package *spotforecast2-safe* (Bartz-Beielstein 2026b), a corrupt load value published by the European Network of Transmission System Operators for Electricity (ENTSO-E) once truncated a leaderboard submission. The fix is stored in the code. But the code does not keep the reasons. Although we tried to keep track of the knowledge from bug fixes in an extra maintained project knowledge base (KB) and in the changelogs, most of it lives in the memory of its maintainer or is simply forgotten. Fowler (2027, ch. 1) argues that this process was rational as long as writing working code was expensive. The code and its documentation were the only reliable memory of the programs. Today, in the age of “Software 3.0” (Karpathy 2025), when an agent can rewrite a module in seconds, the cost structure inverts: implementation becomes cheap and can be quickly repeated. The scarce asset is the knowledge around it: what a module must do, why it does it that way, which incident taught that, and who breaks when it changes. Fowler calls a system built around that knowledge *regenerative*. He proposes the *deletion test* to check whether a module is regenerative:

Pick an important module in your system. If you deleted the implementation tomorrow, could you rebuild the system correctly? Could you reproduce the edge cases, the failure behavior, the timing assumptions, the integration contracts, the operational constraints, the weird exception added after the outage everyone remembers but nobody documented? Could you tell which parts of the current implementation are essential and which are accidental, evaluate a replacement without relying on the old code as the oracle, and prove, with evidence, that the new implementation keeps the promises the system is supposed to keep?

— Fowler (2027, ch. 1)

For most systems, Fowler argues, the honest answer is no. We read his question as three:

- (QF-1) Is the knowledge complete?
- (QF-2) Does a rebuild keep the promises?
- (QF-3) Can a replacement be judged?

The answer of these questions is crucial especially for safety-critical systems like `spotforecast2-safe`. The package is a dependency-minimised port of the forecasting core of `spotforecast2` (Bartz-Beielstein 2026a) and of parts of `skforecast` (Amat Rodrigo and Escobar Ortiz 2026), built for environments in which the EU AI Act (European Parliament and Council 2024) and fail-safe behaviour matter. Its guarantees are determinism, explicit errors instead of silent repair, and a minimal dependency surface, which were defined as *code-development* and *process rules*, see Bartz-Beielstein and Bartz (2026). Its consumers include the sibling open-source package `spotforecast2`, a series of lecture and research books, and daily scripts that submit forecasts to a public ENTSO-E leaderboard. A refactoring that drops one of these guarantees without anybody noticing is exactly the failure a safety case is meant to exclude.

This report describes how the seven primitives of Fowler (2027) (intent, compilation, evaluation, provenance, pace, deletion, and compaction) were applied to `spotforecast2-safe` and what the application produced. The research question is the deletion test itself: can the knowledge that a module needs in order to survive its own deletion be made explicit, machine-checked, and small enough to maintain? The contributions of this report are the following.

1. A mapping of each of the seven primitives to a concrete, versioned artefact of a Python repository, together with a static check that enforces the mapping before every release (Section 2).
2. A pilot on four high-risk modules, with measurements of intent coverage, mutation score, and specification size (Definition 2.31, Definition 2.32, Definition 2.33; Section 3).
3. A regeneration of one module by a large language model (LLM) from its specification alone and a behaviour change carried out specification-first, which together show the method finding and closing real gaps (Section 3.4).
4. Executable examples using a new software architecture completely implemented with local models for every primitive. The examples are cells of this document that run `spotregen`, so a reader can repeat each one.
5. Last but not least, the code used here is available as the open-source Python package `spotregen` on PyPI (Bartz-Beielstein 2026c), which does provide a graphical user interface.

The approach was first tried in the tabular foundation-model project `macl2l` and then generalised for this library. That trial included a first regeneration of one module. This report concentrates on `spotforecast2-safe`, where the stakes are higher because other repositories consume the library’s behaviour.

This report is structured as follows. Section 2 describes the methods and the artefacts that implement them. Section 3 presents the results of the deletion test, including the regeneration and a specification-first change. Section 4 discusses the results and their implications for regenerative software. Section 5 presents the conclusions.

2 Methods

Before methods are described in detail, we give an overview of the terminology used in this report.

2.1 Overview: Terminology used in this report

The *primitives* of Fowler (2027) name the kinds of knowledge. His *durable evaluations* judge every implementation. The remaining terms are ours: the *records* hold the knowledge, the *static check* compares the records with the code, the *protocols* test whether a unit can be rebuilt from the records alone, and the *measures* quantify the result for each pilot unit. Figure 1 places the two kinds of check next to the layers of Fowler’s evaluation suite, Figure 2 shows the relations between all terms, and Table 2 names the `spotregen` subcommand and function behind each term.

Definition 2.1 (Record). A *record* is a machine-readable artefact under `regen/` that holds knowledge about the program and contains no implementation. In `spotforecast2-safe`, the records are the architecture file, one unit record per unit, and the decision records. The archived runs under `regen/runs/` are evidence of past regenerations, not records.

Definition 2.2 (Static check). A *static check* compares the records in `regen/` with the source tree and reports every disagreement as a finding with a check ID, see Table 1. It parses files and runs nothing, so it is deterministic. It stops a change without repairing it. In this report the static check is `spotregen check`. The durable evaluations judge whether a module keeps its promises; the static check judges whether the promises, the architecture, and the

provenance are recorded and still agree with the code. It can only report what disagrees, not what is missing: a comparison of the records with the code cannot see what was never written down.

Table 1: Check IDs of `spotregen` check, grouped by the kind of check. The checkgroups are not the seven primitives. Table 15 relates the two.

checkgroup	check IDs	what is checked
Compilation	ARCH00–ARCH05	parseable sources, allowed imports between subpackages, new or vanished import cycles
Unit	UNIT01–UNIT06	the unit’s file exists, its public names are defined, its imports match <code>depends_on</code> , it declares at least one intent item
Evaluation	EVAL01–EVAL05	evaluations exist, are marked <code>regen</code> , import only public names, cover every intent item
Provenance	PROV01–PROV05	cited decisions exist, evidence resolves, review dates, superseded or orphaned records
Deletion	DEL01–DEL03	every dependent inside the repository is a declared consumer, declared consumers exist

Two kinds of check appear throughout this report. Fowler (2027, ch. 5) supplies the frame for both. His *evaluation suite* has three layers. The bottom layer is *ephemeral*: the ordinary tests of the current implementation, which are regenerated with the code. The middle layer is the *durable evaluations*, which describe behaviour at a boundary and survive a regeneration; run offline against a candidate, they are what he calls the gate that a regenerated component must clear before it ships. The top layer is *live*: online evaluation against production. This report keeps his names for the layers, calls the ephemeral tests *unit tests* (Definition 2.3), and adds one check upstream of the suite, the *static check*, which does not judge a module but the records that describe it.

Definition 2.3 (Unit test). A *unit test* is a test of the current implementation of a unit. It may reach into a private helper or pin an implementation detail, it is regenerated with the code, it is listed in the field `unit_tests` of the unit record (Table 4) and never in the bundle, and it decides nothing about a rebuild. It is the ephemeral layer of Fowler’s evaluation suite (Figure 1). In the regeneration results the old unit tests run after the evaluations as a *second opinion*: a failing old unit test points at an implementation detail or at a promise the specification never recorded (Definition 2.29).

The name *unit test* for the ephemeral layer is ours, as is *evaluation test* for a durable evaluation in pytest form. Fowler speaks of evaluations and of ordinary tests and tells them apart with one question: “Would this assertion still be meaningful if we replaced the implementation tomorrow?” If not, “it may still be a useful test. But it isn’t an evaluation” (Fowler 2027, ch. 3). Importing only public names is our operational answer to that question. It is necessary but not sufficient, since Fowler’s stricter test asks whether “reimplementing the component in a different language would invalidate the evaluation” (Fowler 2027, ch. 5).

Fowler’s point about the evaluations holds for both checks: the generator may be probabilistic, the checks around it are not (Fowler 2027, ch. 4). So does his warning:

The gate is only ever as honest as the fixture beneath it, and a fixture that is thin, stale, or quietly pointed at the wrong boundary lies with total confidence.

— Fowler (2027, ch. 5)

Definition 2.4 (Durable evaluation). A *durable evaluation* is a test that checks whether any implementation of a unit keeps one promise of its intent. It is durable in Fowler’s sense: reimplementing the unit in another language would not invalidate it (Fowler 2027, ch. 5). It is the middle layer of his evaluation suite (Figure 1) and the only judge of a rebuild.

Example 2.1 (Durable evaluations of the pilot units). A unit test (Definition 2.3) belongs to the ephemeral layer of Figure 1: it checks the current implementation, is regenerated with it, and may fail on a correct rebuild. The durable evaluations of the pilot take several forms:

1. An example-based evaluation (Fowler 2027, ch. 5) fixes one input and the required outcome, for instance `test_forbidden_column_in_training_frame_raises` for the leakage guard.
2. A contract evaluation (Fowler 2027, ch. 3, ch. 5) pins what consumers rely on, such as the stable prefix of an error message (`test_unreadable_features_message_has_a_stable_prefix`) or the public signatures and defaults of the outlier detector (`test_public_signatures_and_defaults`).

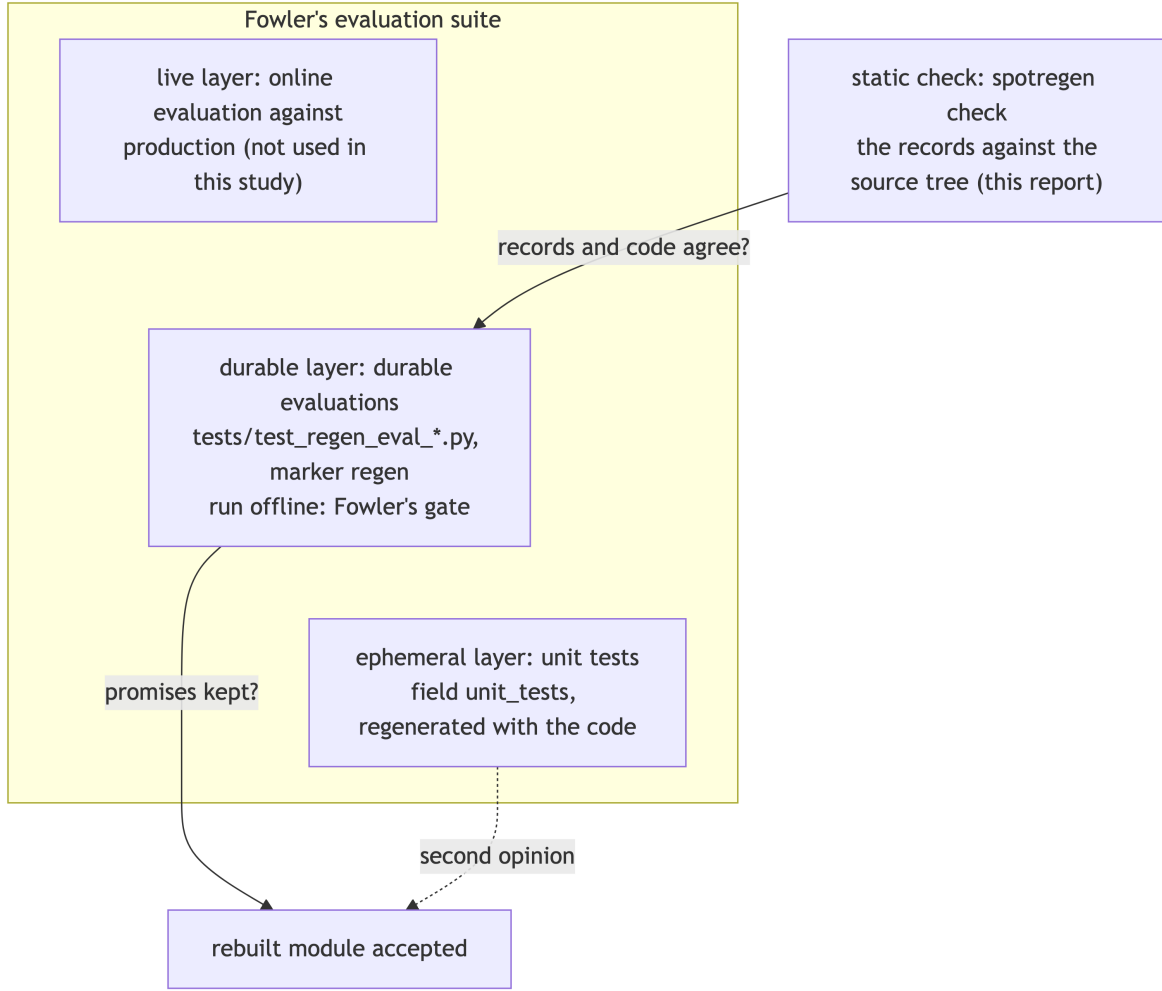


Figure 1: The three layers of Fowler's evaluation suite (Fowler 2027, ch. 5) and the static check of this report. The layers and the gate are Fowler's; the name unit tests for the ephemeral layer and the static check are this report's. A solid edge is a judgement, the dashed edge a second opinion that decides nothing.

3. A property evaluation (Fowler 2027, ch. 3) asserts an invariant of every call, for instance that the guard does not change its inputs (`test_inputs_are_not_mutated`) or that a fixed seed gives the same mask (`test_mask_is_deterministic_for_fixed_seed`).
4. A metamorphic evaluation (Fowler 2027, ch. 5) asserts a metamorphic relation (Definition 2.27) between the outputs of related inputs, for instance that the outlier mask does not change when the data are scaled by a power of two (`test_mask_is_invariant_to_power_of_two_scaling`).
5. A regression evaluation (Fowler 2027, ch. 3, ch. 5) replays a recorded case, such as the flag positions that `spotforecast2-safe 27.0.0` produced on a fixed audit fixture (`test_flag_positions_match_the_27_0_0_audit_fixture`).

Online evaluation, which scores a running system on live traffic (Fowler 2027, ch. 5), is not durable in this sense and is not used.

Definition 2.5 (Protocol). A *protocol* is a written procedure of this study with a fixed order of steps, a fixed judge and a fixed record of the outcome, so that another person can repeat it.

Two protocols are experiments on the records of a unit. Regeneration (Definition 2.29) tests whether the knowledge is complete (QF-1) and whether a rebuild keeps the promises (QF-2). Mutation analysis (Definition 2.28) tests whether the evaluations are strong enough to notice a wrong implementation (QF-3). The third protocol, the specification-first

change (Definition 2.30), is a rule for changing a unit that keeps its records complete. All three protocols use the durable evaluations as judge and none is part of the evaluation suite (Figure 1). A successful run is evidence, not proof: it shows that one rebuild succeeded, not that every rebuild would. In `spotregen`, the field `protocol` of a run record names the procedure that produced the run (Table 2).

Example 2.2 (The checks before a release). `spotforecast2-safe` has no continuous integration (CI), so its checks are make targets that run on the maintainer’s machine before every release. `make verify`, the pre-release verification, runs `make guard`, the linters, the full test suite, and the documentation build, in this order. It stops at the first failure. Two of its steps are the checks of Figure 1:

1. The static check `spotregen check`, the first step of `make guard`, compares the TOML files with the source tree (Section 2.4). One new import in the scoring module fails it with sixty-five findings (Section 3.2). A unit record drafted by `spotregen` fails it with UNIT06 until a person has added an intent item (Section 2.5).
2. The durable evaluations run in `make test` together with the unit tests. They are also the judge of a regeneration: a rebuilt module counts as a success only if it passes every evaluation of its unit (Section 3.4).

Both checks stop a change. Neither repairs it. The repair is left to the person or the model that made the change.

Figure 2 shows how the key terms used in this report, i.e., *primitives*, *pilot units*, *records*, *static check*, *durable evaluations*, *checkgroups*, *protocols*, and *measures* are related. It has two parts:

1. The solid part is the knowledge layer and its checks, which every change passes. `spotregen check` compares the records with the source tree and runs no test (Section 2.4). The durable evaluations run with the unit tests in `make test`. `make verify` runs both before a release. Checkgroups organise the findings of the static check, whereas primitives organise the knowledge, so the two do not map one to one. Compilation, evaluation, provenance, and deletion each have a checkgroup of their own. Intent is enforced through UNIT06 and EVAL04, pace when the records are loaded, and compaction through the report of `spotregen compact` (Table 15).
2. The dashed part is the experiments. A clean static check shows that the records agree with the code, not that they contain enough knowledge to rebuild the unit without it, or that the evaluations would notice a wrong rebuild.

Figure 3 shows the same terms as UML classes with their multiplicities. Figure 4 shows the order in which a change and an experiment use them.

The protocols (Definition 2.5) are the only tests of the records that run anything. The specification-first change tests nothing itself. It keeps the records complete while the code changes, because no behaviour changes without a change of the records first. Of the three measures, intent coverage (Definition 2.31) and specification size (Definition 2.33) follow from the records alone, while mutation score (Definition 2.32) needs mutation analysis and is therefore computed only on request.

Considering the terminal nodes in Figure 2, we get the following deliverables: From the checkgroups we get the check IDs. From the measures we get the numbers (intent coverage, specification size, mutation score).

Table 2: Implementation of the key terms in `spotregen`. Module and function are relative to the package `spotregen` and are checked against the installed version while this document renders. Exit codes: 2 always means a record could not be loaded.

term	defined in	subcommand	module and function	exit code
primitive	Section 2.3	none	none: fields of the records, enforced as in Table 15	–
record	Definition 2.1, Definition 2.11, Definition 2.16	<code>new-unit</code> ; every subcommand loads them	<code>spec.load_spec</code> , <code>scaffold.scaffold_unit</code>	2 on a malformed record
pilot unit	Definition 2.10	<code>status</code> , <code>deletion-test</code>	<code>reports.status_table</code> , <code>reports.deletion_report</code>	<code>deletion-test</code> : 1 unless the verdict is regenerable

term	defined in	subcommand	module and function	exit code
static check:	Definition 2.2	check	checks.run_checks	1 on an error, with
spotregen				--strict also on
check				a warning
checkgroup	Definition 2.24	check	ARCH: checks.check_architecture; UNIT, EVAL, DEL, PROV01, PROV04: checks.check_unit; PROV: checks.check_decisions	as check
durable	Definition 2.4	none: pytest in	drivers._common.run_pytest	pytest's
evaluation		make test	when a protocol scores a module	
unit test	Definition 2.3	none: pytest in	drivers._common.run_pytest, after	pytest's
		make test	the evaluations, as a second opinion	
<i>protocols</i>				
mutation	Definition 2.28	mutate	mutation.generate_mutants,	1 if a mutant
analysis			mutation.mutation_score	survives
regeneration	Definition 2.29	context,	context.build_context,	1 if the final
		claude, local	drivers.claude.run_claude,	module fails
			drivers.local.run_local; the run record's field protocol names the procedure	
specification-	Definition 2.30	none	none: the maintainer writes the item and the evaluations first, which must fail on the old code, then the code, and the static check and the durable evaluations judge it	–
first change				
<i>measures</i>				
intent	Definition 2.31	measures	measures.unit_measures, property	0
coverage			intent_coverage	
mutation score	Definition 2.32	mutate,	mutation.mutation_score	mutate as above
		measures		
		--mutation-score		
specification	Definition 2.33	measures	measures.unit_measures,	0
size			properties specification_lines and ratio	

A language model takes part in two places, outside the figure: it drafts records and rebuilds a module in a regeneration. Nothing it produces is accepted before it passes the checks and, for safety decisions, the maintainer (Section 2.9).

2.2 Object of study

Definition 2.6 (Module). A *module* is a single unit of a Python program, typically a file containing Python code.

Definition 2.7 (Surface). The *surface* of a module is the set of all its public interfaces, including functions, classes, and variables that are accessible from outside the module.

Definition 2.8 (Knowledge). *Knowledge* is the information and understanding that a module possesses about its own behaviour and the behaviour of its dependencies.

Definition 2.9 (Gap). A *gap* is a difference between the knowledge that a module needs to survive its own deletion and the knowledge that is explicitly documented and maintained.

A gap is *pinned* when the missing knowledge has been written down and is checked. Pinning takes two steps. An intent item (Definition 2.15) states the behaviour that was only implicit in the code. A durable evaluation (Definition 2.26) checks it, so that a module that behaves differently fails the evaluation. Where the behaviour was not yet settled, the maintainer decides it before the item is written (Definition 2.30). A pinned behaviour can still be changed, but only on purpose, by changing the item and its evaluation first. A gap is *open* as long as no evaluation checks the behaviour. The report finds gaps in two ways, by the generator's report in a regeneration (Section 3.4) and by mutants that survive

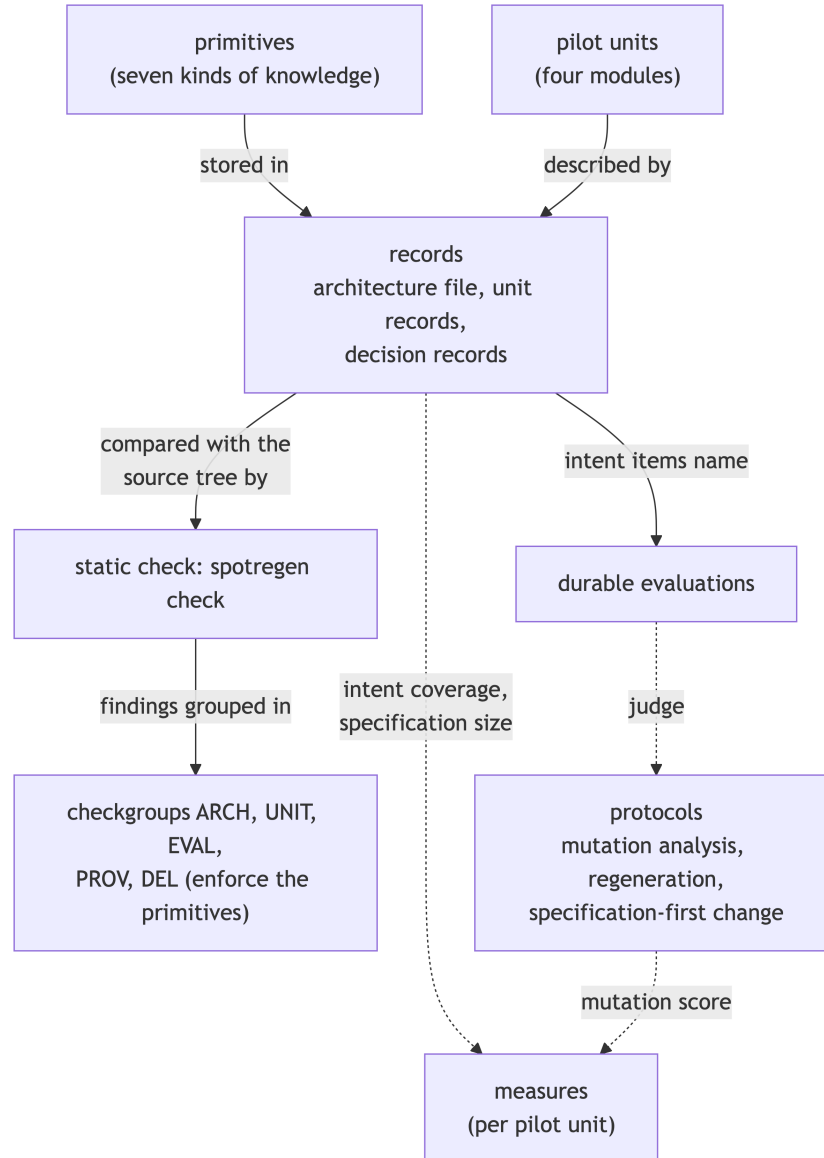


Figure 2: Relations between the key terms of the Methods section. Boxes are terms, edge labels state the relation. Solid edges hold for every change, dashed edges belong to the experiments and the measures. How the checkgroups enforce the primitives is given in Table 15.

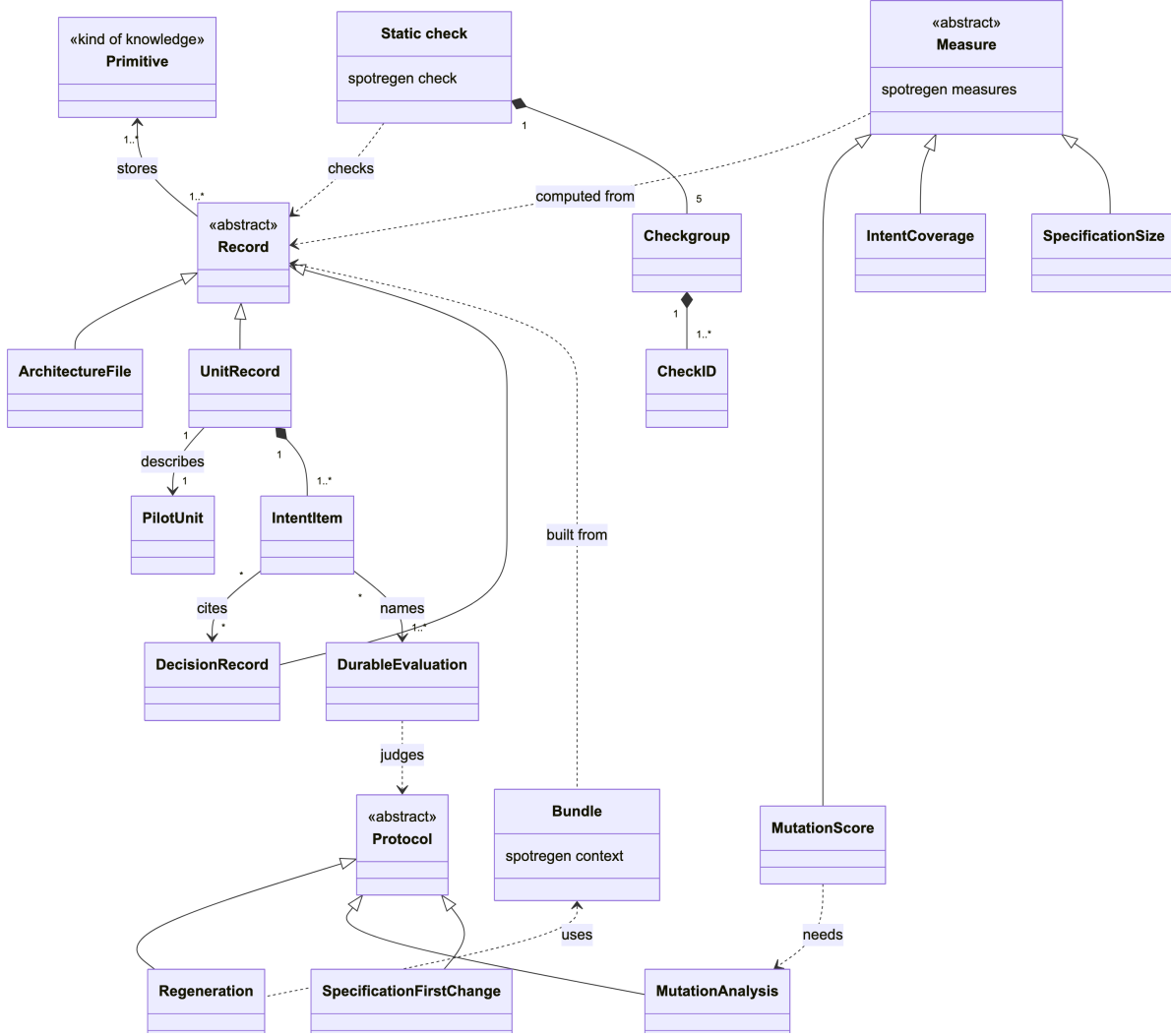


Figure 3: The key terms as a UML class diagram. A hollow triangle marks generalisation, a filled diamond composition, a solid arrow an association with its multiplicities, and a dashed arrow a dependency. The second compartment names the spotregen subcommand behind a class.

the evaluations (Definition 2.28). In the second case a gap counts as pinned when its mutant is killed by a new or strengthened evaluation. The word has a second, common meaning in this report: a pinned version, revision or model is one fixed by its identifier.

Definition 2.10 (Pilot Unit). A *pilot unit* is a unit (Definition 2.11) chosen for the deletion test. It is identified by its unit record, not by its current implementation: the module is what the deletion test removes and rebuilds, the unit record what survives. The four pilot units were selected from the existing modules by the criterion of Fowler (2027, ch. 6) given below.

The study uses spotforecast2-safe version 29.2.1 at commit 41c0e38. The source tree has 19 subpackages and top-level modules, recorded as nodes of the import graph in `regen/architecture.toml`.

Figure 5 shows the parts of the repository that this report uses: the source tree with the four pilot units, the evaluation files, and the knowledge layer under `regen/`.

The `architecture.toml` file names the package and where consumers are searched, lists the imports each subpackage may make and the import cycles accepted as existing debt, and states the verification each pace class requires.

Example 2.3 (The `regen/architecture.toml` file). An excerpt (most entries omitted, long values abbreviated):

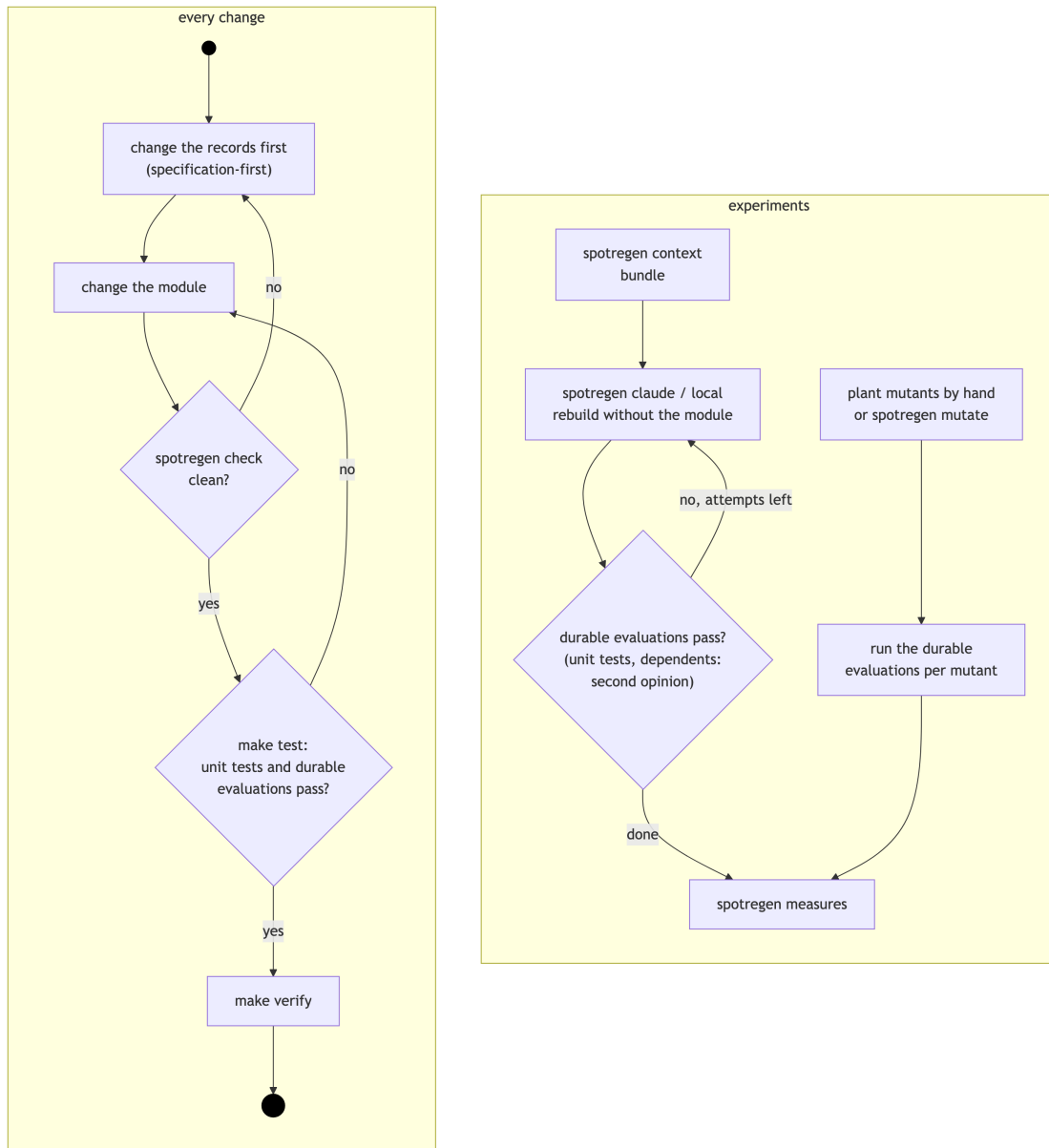


Figure 4: Processes as a UML activity diagram. Filled circles are the initial and the final node, and diamonds are decisions. Left: the path of every change. Right: the experiments, whose two paths are independent. Mermaid has no fork bar, so they are drawn side by side.

spotforecast2-safe/	
└─ src/spotforecast2_safe/	
└─ backtesting/.....	2 modules
└─ calendar/.....	2 modules
└─ configurator/.....	4 modules
└─ data/.....	5 modules
└─ downloader/.....	3 modules
└─ forecaster/.....	11 modules
└─ manager/.....	6 modules
└─ multitask/.....	11 modules
└─ │ guards.py.....	pilot unit
└─ preprocessing/.....	18 modules
└─ │ outlier.py.....	pilot unit
└─ │ target_corruption.py.....	pilot unit
└─ processing/.....	4 modules
└─ │ forecast_scoring.py.....	pilot unit
└─ security/.....	1 module
└─ splitter/.....	5 modules
└─ stats/.....	4 modules
└─ tasks/.....	1 module
└─ utils/.....	4 modules
└─ weather/.....	5 modules
└─ exceptions.py.....	top-level module
└─ submission_cli.py.....	top-level module
└─ tests/	
└─ │ test_regen_eval_forecast_scoring.py.....	evaluations
└─ │ test_regen_eval_guards.py.....	evaluations
└─ │ test_regen_eval_outlier.py.....	evaluations
└─ │ test_regen_eval_target_corruption.py.....	evaluations
└─ regen/	
└─ │ architecture.toml.....	compilation target
└─ │ units/	
└─ │ │ multitask.guards.toml.....	intent, consumers
└─ │ │ preprocessing.outlier.toml.....	intent, consumers
└─ │ │ preprocessing.target_corruption.toml.....	intent, consumers
└─ │ │ processing.forecast_scoring.toml.....	intent, consumers
└─ │ decisions/.....	15 decision records
└─ │ runs/.....	8 archived regeneration runs
└─ HOWTO-REGEN.md.....	repository guide

Figure 5: The parts of the spotforecast2-safe repository used in this report. Subpackages are listed with their number of modules (without `__init__.py`). Of the modules, only the four pilot units are shown, in dark red.

```

package = "spotforecast2_safe"
source_dir = "src"
scan_dirs = ["src", "tests", "scripts", "benchmarks"]
known_cycles = [
    ["__init__", "backtesting"],
    ...
]

[allowed]
exceptions = []
configurator = ["data"]
preprocessing = ["configurator", "data", "exceptions"]
...

[pace]
slow = "human review of the diff and of every intent item it touches, make verify, a
    ↪ MAJOR release if a public contract changes"
fast = "pytest -m regen and the unit's own tests"

```

Four modules were chosen as pilot units, using the criterion that Fowler (2027, ch. 6) gives for where to start and that the repository guide HOWTO-REGEN.md of spotforecast2-safe (Bartz-Beielstein 2026b) adopts:

Start at the boundary where regeneration scares you most: the service you would hesitate to let an agent rewrite. That hesitation is a useful signal. It usually means the implementation is carrying lessons nobody has written down.

— Fowler (2027, ch. 6)

These four modules are as follows:

1. The target-corruption detector (`preprocessing.target_corruption`) guards every daily submission against corrupt ENTSO-E load values and encodes two production incidents.
2. The outlier helpers (`preprocessing.outlier`) carry an earlier architecture decision that made them pure functions.
3. The leakage guard (`multitask.guards`) aborts a submission when a forbidden raw column reaches the model.
4. The scoring functions (`processing.forecast_scoring`) compute the metrics behind the leaderboard reproduction and a published evaluation chapter.

The detector, the guard, and the scoring functions are slow public contracts. The outlier helpers are also used downstream but are classed as medium-pace shared helpers.

Each pilot unit is described by a unit record in `regen/units/`.

Definition 2.11 (Unit record). A *unit* is the part of the program that a unit record describes. It is currently implemented by one module of the source tree, which a regeneration may replace. Its *unit record* is the Tom's Obvious, Minimal Language (TOML) file `regen/units/<unit>.toml` that holds what a regeneration of the module needs to know: the module's name and file (`unit`, `path`), its public contract (`public`), the modules of its own package it may import (`depends_on`), its pace and risk (Section 2.3.5), its intent items with the evaluations that pin them (Section 2.3.1, Section 2.3.3), and its declared consumers (Section 2.3.6). The record describes the module but contains none of its code. The other records under `regen/` are the architecture file `regen/architecture.toml` (Section 2.3.2) and the decision records (Definition 2.16).

2.3 Mapping the seven primitives to artefacts

Fowler (2027) describes the seven primitives in detail. Applying them required a place, a format and a check for each.

Definition 2.12 (Artefact). An *artefact* is a tangible, visible product of the development process. It can be a document, a code file, a test result, or any other deliverable that is produced during the development of a software system.

Before we start, we present a definition of replaceability based on Fowler (2027, ch. 2):

Definition 2.13 (Replaceable). A component is *replaceable* “when you can delete it, regenerate a substitute that honors the same contracts, and deploy it without coordinating changes in other components” (Fowler 2027, ch. 2).

Definition 2.14 (Regeneration Bundle). A *regeneration bundle* is a single Markdown file that `spotregen` context builds for one unit from its records: the architecture file, the unit record, and the decision records it cites. It is the only input for a regeneration and contains no implementation.

Definitions and examples of the seven primitives are as follows.

2.3.1 The Intent Primitive

Definition 2.15 (Intent). Intent “captures what the system must do and why” (Fowler 2027, ch. 3). It includes business rules, constraints, regulatory requirements, timing assumptions, security properties, operational expectations and the reasons behind them. Intent does not describe how the current code happens to work. It describes what any future implementation must preserve.

According to Fowler (2027, ch. 3), an intent item has three parts: the scope (where the requirement applies), the invariant (what must remain true) and the reason (why the invariant matters).

The unit records use these three names as fields. Fowler (2027, ch. 6) also uses the word invariant in his example of a decision record, for what the decision protects. Our decision records keep the field with that meaning (Example 2.6).

Example 2.4 (Intent). On 2026-06-07 the actual load of one bidding zone dropped 11.6 GW below its day-ahead forecast without tripping the corruption detector. The fix added intent item TC-3. Intent items are numbered per unit, with a prefix for the unit, here TC for the target-corruption detector. Its scope is the deviation rule of `detect_target_corruption`. Its invariant is that an hour is flagged when the target stays far enough below the reference for long enough, and that values above the reference never flag. Its reason is that actuals above the forecast are under-forecasting, not corruption.

Example 2.5 (An intent item in a unit record). The unit record `regen/units/multitask.guards.toml` of the leakage guard starts with fields that describe the unit as a whole and continues with its intent items. An excerpt with the header fields and intent item LG-6 (the summary and the other items are omitted):

```
unit = "spotforecast2_safe.multitask.guards"
path = "src/spotforecast2_safe/multitask/guards.py"
pace = "slow"
risk = "high"
public = ["assert_no_leakage"]
evaluations = ["tests/test_regen_eval_guards.py"]
...

[[intent]]
id = "LG-6"
scope = "matching of column names"
invariant = "A column matches only by exact string equality with a forbidden name: no
  ↳ substring, prefix or case-insensitive matching. Derived names such as 'entsoe_
  ↳ forecasted_load' or 'Forecasted Load_lag_24' are not violations of forbidden
  ↳ 'Forecasted Load'. A target column in the training frame is a violation only if the
  ↳ caller put it into the forbidden set."
reason = "Under --entsoe the sanctioned prior enters as the distinctly named entsoe_
  ↳ forecasted_load provider feature; a fuzzy match would abort every --entsoe entry. The
  ↳ training frame carries the target by construction, so the policy side keeps targets
  ↳ out of the forbidden set."
evaluations = ["test_exact_name_match_not_substring", "test_target_column_in_training_
  ↳ frame_trips_only_if_forbidden"]
decisions = ["leakage-guard-three-surfaces"]
...
```

Scope, invariant, and reason are the three parts named above. The field `evaluations` lists the evaluations that check the item, and `decisions` the decision records that justify it.

The most valuable items state what the code must never do, which Fowler (2027, ch. 3) calls *negative constraints*: such constraints “rule out whole classes of implementations while still leaving room for different technical strategies”. They

record exactly what a competent stranger rebuilding the module would not think of. Intent item LG-6 of Example 2.5 is one of them: it forbids matching a column name by substring, prefix or case. Only exact equality counts.

Definition 2.16 (Decision Record). A decision record is a document that captures a decision made during the development of a system, including the context, the decision itself, and the rationale behind it. Decision records extend the architecture decision records of Nygard (2011) with the fields a regenerative system needs.

Example 2.6 (Decision Record). Intent item TC-3 of Example 2.4 rests on the decision record `regen/decisions/target-corruption-deviation-rule.toml`. It states the driver (the incident of 2026-06-07), the decision, the invariant it protects, the evidence, the assumptions, and the rejected alternatives (long fields abbreviated, two of three alternatives omitted):

```
title = "A third, dropout-only deviation rule compares the actual with the published
  ↳ day-ahead forecast"
date = 2026-06-07
status = "active"
driver = "On 2026-06-07 the dropout class stayed below the dynamics thresholds (5.6 GW
  ↳ steps under a 6 GW limit) while the actual sat 11.6 GW under the ..."
decision = "Flag an hour when target - reference < -deviation_mw for at least deviation_
  ↳ slots consecutive native slots (1 on hourly cadence). Asymmetric by ..."
invariant = "The deviation rule never flags a slot where the actual is at or above
  ↳ reference - deviation_mw or where either value is NaN, and never modifies the ..."
origin = "machine"
review_after = 2027-03-31
evidence = ["KB:2026-06-07-deviation-rule-18.1.0", "file:src/spotforecast2_
  ↳ safe/preprocessing/target_corruption.py"]
assumes = ["clean DE deviations reach -8 to -11 GW (q0.001 = -8.3 GW, 2024-01 to
  ↳ 2026-05), so thresholds near 11 GW with slots=1 are the calibrated production ...",
  ↳ ...]

[[alternatives]]
option = "symmetric |target - reference| rule"
rejected_because = "actuals above the forecast are under-forecasting, not corruption"
```

The value `origin = "machine"` marks a record that was reconstructed from the knowledge base after the fact rather than written when the decision was made. Such a record is not a machine-captured fact in the sense of Fowler (2027, ch. 6), which is “cheap to trust”, but a rationale asserted by a model. It carries the least weight until the maintainer confirms it.

2.3.2 The Compilation Primitive

Fowler (2027, ch. 3) argues that intent is necessary but not sufficient to specify a repeatable system. How the intent is implemented technologically and architecturally must be captured as well.

Definition 2.17 (Compilation). Compilation is the architecture that generated code compiles into: boundaries, ownership, communication, runtime, and framework constraints. Humans specify it, so that a generator does not have to infer it from the old implementation (Fowler 2027, ch. 3).

Fowler (2027, ch. 3) lists what a compiled architecture defines, among them where the boundaries are, which failure modes must be handled locally and which implementation choices are left open. As Example 2.3 shows, it is specified in a TOML file, which is written mostly by humans. The static check covers only the boundaries. The conventions, such as fail-safe over silent repair, reach the generator through the bundle and are checked only where an evaluation pins them.

Example 2.7 (Compilation). In `spotforecast2-safe, regen/architecture.toml` lists for every subpackage the subpackages it may import. `processing` may import none, so a single planted import of the `multitask` package into the scoring module fails the static check (Section 3.2). The excerpt shows the two entries of `[allowed]` involved:

```
[allowed]
...
multitask = ["calendar", "configurator", "data", "downloader", "exceptions",
  ↳ "forecaster", "manager", "preprocessing", "processing", "splitter", "weather"]
```

```
processing = []
...
```

The planted line in `processing/forecast_scoring.py` is

```
from spotforecast2_safe.multitask import guards
```

`processing` is not allowed to import `multitask`, so `spotregen` check reports the edge as ARCH01. Table 3 lists the keys of `architecture.toml` that `spotregen` reads. Only `package`, `source_dir` and `[allowed]` must be filled. Example 2.3 shows the values `spotforecast2-safe` uses.

Table 3: Keys of `regen/architecture.toml` (`spotregen` 0.10.1).

Key	Required	Meaning
<code>package</code>	yes	top-level package name
<code>source_dir</code>	yes	directory, relative to the repository root, that holds the package
<code>[allowed]</code>	yes	per subpackage or top-level module, the subpackages it may import; " <code>__init__</code> " is the package's <code>__init__.py</code>
<code>[pace]</code>	no	per pace class, the verification a change at that pace needs
<code>known_cycles</code>	no	accepted import cycles, each a pair of subpackages; any other cycle fails the check (ARCH04)
<code>scan_dirs</code>	no	directories whose Python files count as consumers (default <code>src</code> , <code>tests</code> , <code>scripts</code>)
<code>doc_globs</code>	no	glob patterns of documents whose text counts as a consumer
<code>conventions</code>	no	rules every implementation follows
<code>open_choices</code>	no	implementation choices deliberately left to the generator
<code>decisions</code>	no	ids of the decision records behind the architecture itself
<code>kb_dir</code>	no	directory of knowledge-base entries for KB: evidence

The compilation target records which subpackage may import which. It was recorded as a baseline from the tree of 2026-09-29, together with the import cycles that existed at that time. The cycles are treated as existing debt: a new cycle fails the check, whereas a vanished one must be removed from the list, so the list can only shrink. The module boundaries of Parnas (1972) thereby become a checked artefact instead of a convention.

2.3.3 The Evaluation Primitive

After specifying the architecture, the verification must be done. This is done by “tests that outlive the code” (see Fowler 2027, ch. 3). The tests are called evaluations.

Definition 2.18 (Evaluation). An evaluation is “an executable claim about system behavior that should remain meaningful across implementations” (Fowler 2027, ch. 3). It judges the new code without reference to the old one.

Example 2.8 (Evaluation). The evaluation of intent item TC-2 asserts that the flag mask of the corruption detector does not change when target and reference are shifted by a constant, when values and thresholds are scaled together, or when the index is shifted by whole hours. It needs no list of the corrupt hours and imports only public names.

The unit record links the evaluations at two levels: the header names the evaluation files, each intent item the evaluations that check it. An excerpt of `regen/units/preprocessing.target_corruption.toml` with intent item TC-2 (long values replaced by "..."):

```
...
evaluations = ["tests/test_regen_eval_target_corruption.py"]
unit_tests = [
    "tests/preprocessing/test_target_corruption.py",
    "tests/multitask/test_prepare_data_target_corruption.py",
    "tests/test_config_target_corruption_knobs.py",
]
...
```

```

[[intent]]
id = "TC-2"
scope = "...
invariant = "...
reason = "...
evaluations = [
    "test_step_defect_flags_whole_calendar_hours_on_both_endpoints",
    "test_intra_hour_range_rule_flags_the_hour",
    "test_corruption_older_than_window_is_ignored",
    "test_rules_compose_by_union_and_thresholds_are_monotone",
    "test_mask_is_invariant_to_level_scale_and_whole_hour_shift",
    "test_flags_are_or_ed_across_target_columns",
    "test_range_and_step_thresholds_are_strict",
]
decisions = ["target-corruption-truncate-whole-tail"]
...

```

The evaluation file marks every test in it as durable, which is what EVAL02 checks:

```
pytestmark = pytest.mark.regen
```

Table 4 lists the fields of a unit record that concern evaluations.

Table 4: Fields of a unit record that concern evaluations. *Not required by the loader, but an intent item without an evaluation fails the static check (EVAL04). An evaluation named by no intent item is reported as EVAL05.

Field	Level	Required	Meaning
evaluations	unit	no	repository-relative paths of the evaluation files; each must exist (EVAL01) and carry the marker <code>regen</code> (EVAL02)
unit_tests	unit	no	the unit tests of the ephemeral layer (Definition 2.3): coupled to the implementation, regenerated with the code, not part of the regeneration bundle
evaluations	intent item	yes*	names of the evaluations that check the item; each must exist in an evaluation file of the unit (EVAL01)

2.3.4 The Provenance Primitive

Definition 2.19 (Fail-closed). A fail-closed component refuses to proceed when it cannot establish that proceeding is safe: it raises an exception instead of logging a warning and continuing in an unknown state. The idea is the principle of fail-safe defaults (Saltzer and Schroeder 1975, sec. I.A.3): a mechanism that permits only under explicitly stated conditions “tends to fail by refusing permission, a safe situation, since it will be quickly detected”, whereas a mechanism that only excludes known bad cases tends to fail by allowing access, “a failure which may go unnoticed in normal use”.

Example 2.9 (Fail-closed Example). In `spotforecast2-safe`, the leakage guard is fail-closed. When it cannot read the model, finds a surface missing, or receives an empty target list, it raises an exception and aborts the submission instead of logging a warning. It has done so since the first regeneration exposed that the old code answered these three cases leniently (Section 3.4.1).

Fail-closed is narrower than fail-safe. A fail-safe component moves to a safe state when it fails. Which state is safe depends on the application: an emergency exit unlocks when the power fails, whereas an access check denies. In `spotforecast2-safe`, fail-safe names the library-wide convention that invalid input raises an explicit exception instead of being repaired silently. Fail-closed fixes the direction and widens the trigger. The leakage guard stops not only when it detects a leak but also when it cannot decide whether there is one, so an unknown state counts as unsafe. Both behaviours are defaults. A user who sets `on_weather_failure="skip"` switches the fail-safe behaviour off for Open-Meteo failures: the run logs a warning and continues without weather features. The responsibility for that forecast passes to the user.

Definition 2.20 (Provenance). Provenance “records the derivational history of a component: what intent produced it, which architecture it compiled into, what evaluations validated it, what evidence influenced it, and what decisions changed it” (Fowler 2027, ch. 3).

Example 2.10 (Provenance). The decision record `leakage-guard-fail-closed` of 2026-09-30 states why the leakage guard raises when it cannot read a surface: the first regeneration had exposed three lenient gaps. The record names its origin (the maintainer) and a review date, 2027-09-30, after which the static check asks whether the rationale still holds (PROV03).

Example 2.6 shows a whole decision record. The excerpt below of `regen/decisions/leakage-guard-fail-closed.toml` keeps only the fields that carry its provenance: when it was taken, by whom, on what evidence, under which assumptions, and until when it holds (title, driver, decision, invariant, and alternatives omitted):

```
...
date = 2026-09-30
status = "active"
approved_by = "Thomas Bartz-Beielstein, 2026-09-30 (answer to the open safety decisions
↪ of the regeneration report)"
origin = "human"
evidence = [
  "KB:2026-09-29-regenerative-software-implemented",
  "file:src/spotforecast2_safe/multitask/guards.py",
  "file:tests/test_regen_eval_guards.py",
]
assumes = [
  "consumers catch LeakageError before RuntimeError (abort vs warn-and-skip); a new
  ↪ RuntimeError case therefore degrades to a logged warning in the submitters, never
  ↪ to a crash",
  "BaseTask initialises exog_feature_names to [] and sets data_with_exog in prepare_
  ↪ data, so a trained pipeline never has these surfaces missing",
]
review_after = 2027-09-30
...
```

The first assumption no longer holds. Since 2026-10-02 the submitters abort on a `RuntimeError` as well. The later record `leakage-guard-message-contract` states that it supersedes this assumption (Section 3.4.7). Table 5 lists the fields of a decision record and the checks of `spotregen` check that use them.

Table 5: Fields of a decision record in `regen/decisions/` (spotregen 0.10.1).

Field	Required	Meaning
<code>title, date</code>	yes	what was decided, and when
<code>driver</code>	yes	the problem or incident that forced the decision
<code>decision</code>	yes	what was decided
<code>approved_by</code>	yes	the person who approved it, with date and occasion
<code>status</code>	no	active (default), superseded or expired; an intent item that rests on a record that is not active fails the check (PROV04)
<code>origin</code>	no	human (default) or machine, for a record reconstructed by a model
<code>invariant</code>	no	what the decision protects
<code>evidence</code>	no	entries starting with <code>file:</code> , <code>commit:</code> , <code>KB:</code> or <code>url:</code> ; each must resolve (PROV02)
<code>assumes</code>	no	conditions under which the decision stays valid
<code>review_after</code>	no	date after which an active record is reported as due for review (PROV03)
<code>[[alternatives]]</code>	no	option (required) and <code>rejected_because</code> , one table per rejected alternative

Besides the driver, the decision, and the rejected alternatives, a decision record lists machine-resolvable evidence (a file, a commit or a KB entry), the assumptions it rests on, a review date, and an `origin` that says whether a person

made the decision or whether it was reconstructed from code and history. The review date addresses the failure mode that Fowler (2027, ch. 6) describes as provenance that lies, “records that are confidently wrong”: a rationale that was true two regenerations ago and quietly stopped being true misleads worse than a missing one. A review date is only a proxy. Fowler (2027, ch. 6) ties a record to “the conditions it assumed”, so that the system can flag it when they change. The static check shows the assumes list once the review date has passed, but it cannot notice that an assumption broke earlier.

2.3.5 The Pace Primitive

Definition 2.21 (Pace). Pace matches the speed at which a component may be replaced to its position in the architecture. Foundational layers change more slowly, and under more verification, than experiments (Fowler 2027, ch. 3).

Example 2.11 (Pace). The leakage guard is a slow unit. A change to it requires a human review of the diff and of every intent item it touches, make verify, and a major release if a public contract changes. The medium-pace outlier helpers need only the tests, the evaluations, and a review of the contract diff.

Pace is recorded in two places. `regen/architecture.toml` states in `[pace]` what each class requires:

```
...
[pace]
slow = "human review of the diff and of every intent item it touches, make verify, a
  ↳ MAJOR release if a public contract changes; downstream (spotforecast2, bart26*)
  ↳ relocked and re-run"
medium = "make test and pytest -m regen, review of the public contract diff"
fast = "pytest -m regen and the unit's own tests"
```

Each unit record names its class in pace, next to risk. The leakage guard and the outlier helpers:

```
# regen/units/multitask.guards.toml
...
pace = "slow"
risk = "high"
...
# regen/units/preprocessing.outlier.toml
...
pace = "medium"
risk = "high"
...
```

Table 6 lists the fields. `spotregen` checks that every value belongs to its vocabulary and stops with exit code 2 otherwise. It does not run the verification a class requires: it copies the requirement of the unit’s class into the regeneration bundle and shows the units by class in `spotregen status`. The maintainer carries it out.

Table 6: Fields that record the pace of a unit (spotregen 0.10.1).

Field	File	Required	Meaning
<code>[pace]</code>	<code>architecture.toml</code>	no	per class fast, medium or slow, the verification a change at that pace needs
<code>pace</code>	unit record	yes	the class of the unit: fast, medium or slow
<code>risk</code>	unit record	yes	low, medium or high: how much verification a regeneration of the unit needs

The pace of a unit takes up the shearing layers of Brand (1994), the parts of a building that change at different rates. Fowler (2027, ch. 3) applies the idea to software as fast and slow layers: not every layer should change at the same speed. It is the layer, not the size of the diff, that decides how dangerous a change is. Each unit carries one of three pace classes. `architecture.toml` states the verification each class requires.

2.3.6 The Deletion Primitive

Definition 2.22 (Deletion). Deletion makes it safe to remove or replace a component without breaking hidden dependencies (Fowler 2027, ch. 3). Its diagnostic, the operational replacement test that we call the *deletion test*, asks “could

we delete this and regenerate it by Friday? If not, what’s in the way?” and is meant to be run, not argued about: “The attempt itself is the instrument” (Fowler 2027, ch. 2). In this study the deletion test of a unit has four components:

1. *Blast radius*. Who would notice that the unit is gone? Its dependents are declared as consumers.
2. *Knowledge outside the implementation*. Every intent item names evaluations. The decisions behind it are on record.
3. *Rebuild*. The implementation is deleted and rebuilt from that knowledge alone, by regeneration (Definition 2.29).
4. *Judgement*. The rebuild is judged by the evaluations, without the old code as the oracle. Mutation analysis (Definition 2.28) shows that the evaluations would notice a wrong rebuild.

The subcommand `spotregen deletion-test` checks components 1 and 2 before any deletion and deletes nothing, so its verdict “regenerable” is a precondition for an attempt, not its result. Regeneration carries out components 3 and 4. It also runs the dependents inside the repository against the rebuild. Dependents in other repositories are run in a separate staging step (Section 3.4.7).

Example 2.12 (Deletion). The deletion test of the leakage guard covers all four components.

1. *Blast radius*. `spotregen deletion-test` lists the guard’s dependents: submitter scripts in four downstream repositories, lecture chapters, a challenge paper, the daily submission runs and the operator who reads its messages. Most of them lie outside the repository, so they must be declared as consumers (Definition 2.22). The consumers that execute the guard were also run against the rebuilds (Section 3.4.7).
2. *Knowledge outside the implementation*. The same report lists the 8 intent items LG-1 to LG-8, each with its evaluations and decision records. It ends with the verdict “regenerable”.
3. *Rebuild*. The guard was deleted in a separate worktree and rebuilt from its bundle, by Claude Opus 5.5 (Section 3.4, Section 3.4.2) and by local models (Section 3.4.4).
4. *Judgement*. Every rebuild was scored against the evaluations of the guard as of its run (44 today), with the old unit tests as a second opinion. All mutants planted in the guard were killed (Section 3.1), so a wrong rebuild would have been noticed.

The blast radius of component 1 is declared in the unit record as a list of `[[consumers]]` tables. An excerpt of `regen/units/multitask.guards.toml` with one consumer of each kind the guard uses (the unit record declares 22 consumers in all):

```
...
[[consumers]]
ref = "src/spotforecast2_safe/multitask/__init__.py"
kind = "code"
note = "re-exports assert_no_leakage as spotforecast2_safe.multitask.assert_no_leakage
↳ (in __all__)"

[[consumers]]
ref = "tests/test_multitask_guards.py"
kind = "test"
note = "implementation-coupled unit tests (also import the private _fitted_feature_
↳ names)"

[[consumers]]
ref = "bart261-vorlesung:scripts/_team4_submit_common.py"
kind = "external"
note = "shared team4 submitter: forbidden {'Forecasted Load', 'Actual'}; LeakageError ->
↳ Abort(1, str(exc)), other RuntimeError -> Abort(1, 'leakage guard could not verify
↳ the model: ...') since 2026-10-02"

[[consumers]]
ref = "daily ENTSO-E leaderboard submissions of the team4 entries"
kind = "operational"
note = "a LeakageError or RuntimeError aborts the submit run with exit code 1 before
↳ write_submission; audit log records the message"
```

```
[[consumers]]
ref = "Thomas Bartz-Beielstein"
kind = "human"
note = "operator of the submit runs; reads the abort message: a leak (task, surface,
  ↪ columns) or an unverifiable model (task, target, cause)"

[[consumers]]
ref = "docs/regenerative.qmd"
kind = "doc"
note = "pilot table and the regeneration experiment of the Regenerative Software page"
...
```

Table 7 lists the fields of a consumer. The static check resolves the consumers inside the repository and leaves the others to the people who read the unit record.

Table 7: Fields of a `[[consumers]]` table in a unit record (spotregen 0.10.1). A path of the five repository kinds must exist (DEL02) and must still reference the unit (DEL03). A file that references the unit without being declared is reported as DEL01. Consumers of the kinds `external`, `operational` and `human` are not checked.

Field	Required	Meaning
<code>ref</code>	yes	what depends on the unit: a repository path for code, test, script, doc and book; <code><repo>:<path></code> for <code>external</code> ; a description for <code>operational</code> and <code>human</code>
<code>kind</code>	yes	one of code, test, script, doc, book, external, operational, human
<code>note</code>	no	how the consumer depends on the unit, for example how it handles the unit's errors

A consumer is a declared dependent of a unit: something that would notice its deletion. Fowler (2027, ch. 2) speaks of dependents and distinguishes direct, indirect, and environmental ones. We keep the field name `consumers` of the records. Consumers are declared per unit with a `kind` (code, test, script, doc, book, external, operational, human). The `external` kind names a file in a downstream repository as `<repo>:<path>`. It cannot be checked from inside the repository, but it makes the blast radius of a change visible before the change is made.

2.3.7 The Compaction Primitive

Definition 2.23 (Compaction). Compaction is “the discipline that prevents regeneration from becoming accumulation” (Fowler 2027, ch. 3). It removes what earlier generations left behind. Fowler applies it to interfaces, configuration, dependencies, evaluations and, above all, concepts. In this study it covers only the records in `regen/`, so it finds stale knowledge, not obsolete paths of the library itself.

Example 2.13 (Compaction). A unit record that still declares a dependency its code no longer imports is sediment. `spotregen compact` lists it as a candidate for removal.

Compaction has no fields of its own. It reads fields of the other records and reports those that no longer match the code. The outlier helpers import nothing else from the package. Their unit record says so:

```
# regen/units/preprocessing.outlier.toml
...
depends_on = []
...
```

Sediment arises when this line is changed to

```
depends_on = ["spotforecast2_safe.utils"]
```

The module still does not import `utils`, so the entry promises a dependency that does not exist. `spotregen compact` reports it as UNIT05. Table 8 lists the fields that compaction reads and the sediment it looks for.

Table 8: Fields read by compaction (spotregen 0.10.1). All these checks are warnings: they ask for a removal or a review and do not fail the check unless `--strict` is set. `spotregen compact` collects all of them except ARCH05, which `spotregen check` reports.

Field	File	Sediment	Check ID
<code>depends_on</code>	unit record	a module that is declared but no longer imported	UNIT05
<code>[[consumers]] ref</code>	unit record	a consumer in the repository that no longer references the unit	DEL03
<code>[[intent]] evaluations</code>	unit record	an evaluation in an evaluation file that no intent item names	EVAL05
<code>[[intent]] decisions</code>	unit record	an intent item that rests on a decision that is not active	PROV04
—	decision record	a decision that no intent item and no architecture entry cites	PROV05
<code>review_after</code>	decision record	an active decision past its review date	PROV03
<code>known_cycles</code>	<code>architecture.toml</code>	an accepted import cycle that no longer exists	ARCH05

Table 9 shows the mapping. All knowledge is stored as TOML files in a tracked top-level directory `regen/`, so it is reviewed, diffed, and versioned like code.

Table 9: The seven primitives of Fowler (2027) and where each one lives in the repository.

primitive	question it answers	artefact in spotforecast2-safe
Intent	What must stay true, and why?	<code>regen/units/<unit>.toml</code> , <code>[[intent]]</code> items
Compilation	Which architecture must the code fit into?	<code>regen/architecture.toml</code>
Evaluation	How is a replacement judged?	<code>tests/test_regen_eval_<unit>.py</code> , marker <code>regen</code>
Provenance	Why is the system this way?	<code>regen/decisions/<id>.toml</code>
Pace	How fast may this layer change?	pace of each unit, policy in <code>[pace]</code>
Deletion	Who notices if it is gone?	<code>[[consumers]]</code> of each unit
Compaction	What can be removed?	<code>spotregen compact</code> , <code>known_cycles</code>

2.4 The static check

The primitives are enforced by `spotregen` (Bartz-Beielstein 2026d), a standard-library package without runtime dependencies. The study began with `spotregen` as a script, `scripts/regen.py`, inside `spotforecast2-safe`. On 2026-10-01 it was released as its own package, so that `macl2l` and this report can share one version. `spotforecast2-safe` takes it as a development dependency only. It is not part of the distributed wheel, so the safety-critical package, its software bill of materials (SBOM), and the scope of its model card are unchanged. The subcommand `check` is the static check (Definition 2.2). The durable evaluations in `make test` are the judge of an implementation. `check` runs as the first step of `make guard` and therefore of `make verify`, the pre-release verification of the project, which has no CI.

The static check compares the TOML files with the source tree and reports every disagreement as a finding that carries the stable *check ID* of the check that found it. The check IDs are organised in checkgroups.

Definition 2.24 (Checkgroup). A *checkgroup* is the set of checks of `spotregen check` whose check IDs share a prefix. Each checkgroup asks one kind of question about the records in `regen/`. The check ID ARCH asks whether the source tree keeps to the architecture file, UNIT whether a unit’s module matches its unit record (Definition 2.11), and EVAL whether the unit’s evaluations exist and cover its intent items. PROV asks whether the decision records are cited, their evidence resolves, and their reviews are current. DEL asks whether every dependent of a unit is a declared consumer. A checkgroup is named after the primitive it serves, where it serves one. It groups check IDs and is not a unit of `spotregen`. All five are implemented in the single unit `spotregen.checks`, whose function `check_unit` reports UNIT, EVAL, and DEL findings together with PROV01 and PROV04. A primitive, in contrast, is a kind of knowledge (Section 2.3), so checkgroups and primitives need not correspond one to one.

A check ID consists of the prefix of its checkgroup and a two-digit number. Each check has a fixed severity. Errors fail the check. Warnings ask for attention. With `--strict`, warnings fail the check as well. Table 1 summarises the five checkgroups. Table 10 to Table 14 list every check ID.

Table 10: Check IDs of the Compilation checkgroup, read from `regen/architecture.toml` and the import graph.

check ID	severity	reported when
ARCH00	error	a source file of the package does not parse
ARCH01	error	a subpackage imports a subpackage that [allowed] does not permit
ARCH02	error	a subpackage is missing from [allowed]
ARCH03	warning	an allowed edge is unused, or a listed subpackage does not exist
ARCH04	error	a new import cycle, not listed in <code>known_cycles</code>
ARCH05	warning	a cycle listed in <code>known_cycles</code> no longer exists

Table 11: Check IDs of the Unit checkgroup, checking a unit record against its module.

check ID	severity	reported when
UNIT01	error	the unit's module does not exist or does not parse
UNIT02	error	path is not the file of the module the unit record names
UNIT03	error	a name in <code>public</code> is not defined in the module
UNIT04	error	the module imports a package module that is not in <code>depends_on</code>
UNIT05	warning	an entry of <code>depends_on</code> is not imported
UNIT06	error	the unit record declares no intent item

Table 12: Check IDs of the Evaluation checkgroup, checking the evaluation files and their links to intent items.

check ID	severity	reported when
EVAL01	error	an evaluation file or a named evaluation does not exist, or the file does not parse
EVAL02	error	an evaluation file lacks the pytest marker <code>regen</code>
EVAL03	error	an evaluation imports a name of the unit that is not <code>public</code>
EVAL04	error	an intent item names no evaluation
EVAL05	warning	an evaluation is not linked to any intent item

Table 13: Check IDs of the Provenance checkgroup, checking the decision records and the citations of them.

check ID	severity	reported when
PROV01	error	an intent item or the architecture cites an unknown decision
PROV02	error	evidence of a decision does not resolve (file, commit, KB entry or prefix)
PROV03	warning	an active decision is past its <code>review_after</code> date
PROV04	warning	an intent item rests on a decision that is not active
PROV05	warning	no intent item or architecture entry cites the decision

Table 14: Check IDs of the Deletion checkgroup, checking the declared consumers of a unit.

check ID	severity	reported when
DEL01	error	a file in the repository depends on the unit but is not a declared consumer
DEL02	error	a declared consumer in the repository does not exist
DEL03	warning	a declared consumer no longer references the unit

By Definition 2.24, the five checkgroups are not the seven primitives. Four checkgroups serve one primitive each:

- ARCH serves Compilation,
- EVAL serves Evaluation,
- PROV serves Provenance and
- DEL serves Deletion.

The Unit checkgroup is not a primitive. It checks that a unit record matches its module (UNIT01 to UNIT03), which every other check of the unit presupposes. It serves two primitives:

- Compilation, because the imports of the module must match `depends_on` (UNIT04, UNIT05), and
- Intent, because a unit must state at least one intent item (UNIT06).

The remaining three primitives have no checkgroup of their own, for different reasons summarised in Table 15:

- Intent is enforced through other checkgroups: UNIT06 requires an intent item, EVAL04 an evaluation for each one.
- Pace is a property of a unit record, not a relation between the unit record and the tree. Its value is checked against the vocabulary `fast`, `medium`, and `slow` when the file is loaded. A wrong value stops `spotregen` with exit code 2 instead of yielding a finding (Section 2.5). The verification each pace requires is stated in the regeneration bundle and applied by the maintainer, not by the static check.
- Compaction is a report rather than a check: `compact` collects the warnings that mark sediment, which are UNIT05, EVAL05, DEL03, and PROV03 to PROV05. A vanished import cycle is reported by `spotregen` check itself as ARCH05.

Table 15: How the static check and the other subcommands enforce each primitive. UNIT01–UNIT03 check that a unit record matches its module and precede all checks of a unit.

primitive	enforced by
Intent	UNIT06 (no intent item), EVAL04 (intent item without evaluation)
Compilation	ARCH00–ARCH05, UNIT04, UNIT05
Evaluation	EVAL01–EVAL05
Provenance	PROV01–PROV05
Pace	vocabulary check on loading (exit code 2); <code>status</code> , regeneration bundle
Deletion	DEL01–DEL03; <code>deletion-test</code>
Compaction	<code>compact</code> (UNIT05, EVAL05, DEL03, PROV03–PROV05); ARCH05 for <code>known_cycles</code>

Four further subcommands support the workflow. `status` lists the units by pace, risk and coverage. `deletion-test` prints the blast radius and the knowledge held outside the implementation of one unit and ends in a verdict. `context` writes the regeneration bundle, the complete input a generator receives. `compact` lists compaction candidates.

2.4.1 The graphical interface

Since version 0.8.0, `spotregen` also has a graphical user interface (GUI), for maintainers who would rather not memorise the subcommands and their options. It is built on `tkinter` from the Python standard library, so the package still has no runtime dependencies. It computes nothing of its own: every value it shows comes from the functions that the subcommands call. Parity tests compare its answers with the output on the command line. The design is recorded as ADR 0002 in the `spotregen` repository. For a given repository, the interface is started with

```
spotregen --root ~/workspace/spotforecast2-safe gui
```

or, equivalently, with `spotregen gui --root DIR`, `spotregen-gui` or `python -m spotregen.gui`. The installed version documents the subcommand as follows:

```
usage: spotregen gui [-h] [--root ROOT]
```

options:

```
-h, --help    show this help message and exit
--root ROOT   repository root (default: search upwards)
```

The window opens with the Overview and runs `check` in the background at once. The Overview answers the three questions a maintainer asks first: does the specification load, does the check pass, and which units have every intent item evaluated? Below the answers it lists the next actions, each with a button to the view that resolves it and the exact command it corresponds to. Figure 6 shows the Overview on the sample repository of `spotregen`, into which three faults were planted: a forbidden import from `core` into `app`, which also closes an import cycle, a script that uses

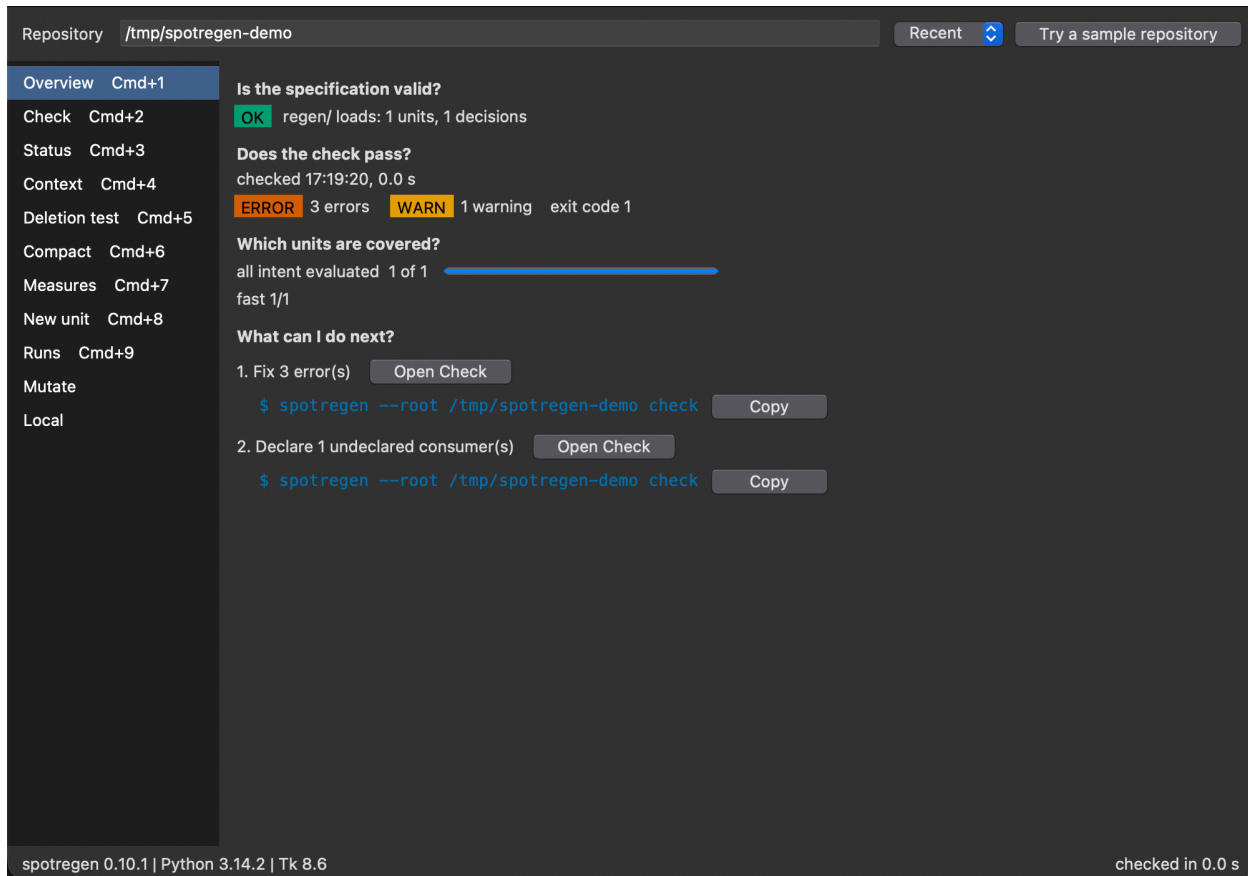


Figure 6: The Overview after the check on the sample repository with three planted faults. Each next action links to the view that resolves it and shows the equivalent command.

the unit without being declared as its consumer, and a declared consumer that no longer mentions the unit. The three faults yield four findings, ARCH01, ARCH04, DEL01, and DEL03.

The Check view (Figure 7) lists the findings with check ID, file, and line. It can filter them by level, check ID, and text. The detail line under the table is the line the command line prints, character for character. The command underneath reproduces the result in a terminal, so the interface also teaches the subcommands. Enter opens the file at the reported line in a viewer, Cmd+E in Vim. The option Strict makes warnings fail the check, as `check --strict` does. It updates the exit code and the command. The Status view shows the output of `status` as a table grouped by pace.

Since version 0.9.0, five further views cover the remaining read-only subcommands. Context shows the regeneration bundle of a unit and saves it under `regen/runs/`, but never over an existing file. Deletion test shows the report of `deletion-test` with its verdict as a badge. Compact lists the compaction candidates. New unit drafts a unit record from a module picker and two choices for pace and risk, shows the draft that `new-unit --dry-run` prints, writes it, and checks the new record at once. Measures shows intent coverage and specification size per unit and starts the mutation score as a background job (Figure 8). The job runs the command line in a separate process, streams its progress, writes a log to the run directory, and can be cancelled. Cancelling sends an interrupt to the whole process group and escalates to termination, so that no mutant run outlives the window. The interface offers no runs of the Claude driver at all.

Version 0.10.0 adds the two long-running subcommands that stay on the maintainer’s machine, together with a view of their results. Mutate measures the mutation score of one unit. Local regenerates a unit with a model that LM Studio serves locally. Both run as background jobs of the kind described above, each in a new run directory `regen/runs/<date>-<unit>-<label>/`. Their forms are generated from the option definitions of the command line. A test fails as soon as the command line gains an option that the form neither shows nor deliberately hides, so the interface cannot fall behind the subcommands. The Runs view (Figure 9) lists the run directories, newest first, with unit, label, protocol, and mutation score. It opens a directory or its `run.toml` and `score.toml`.

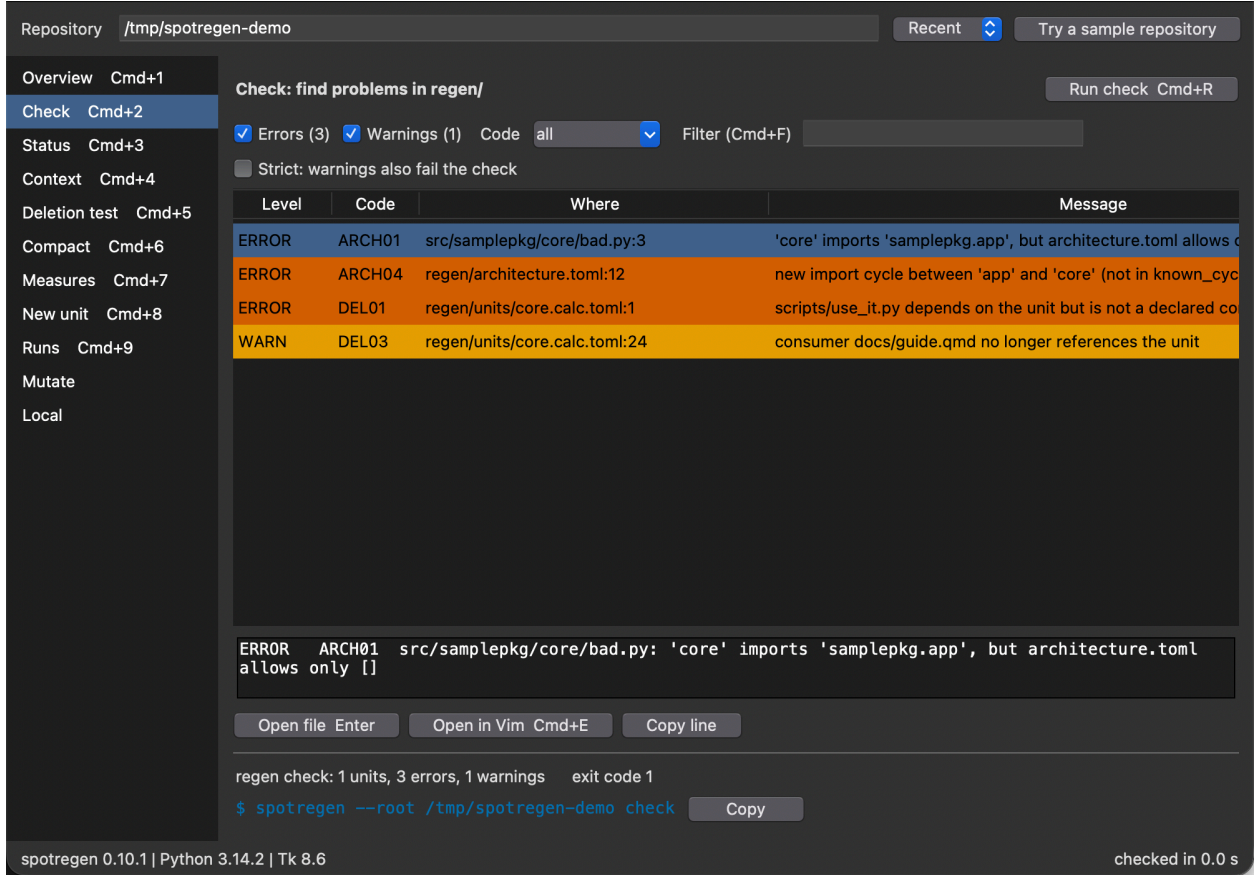


Figure 7: The Check view on the same repository. The selected finding is shown below the table as the command line prints it.

2.5 How to generate and edit the TOML files

The knowledge in regen/ comes in three kinds of TOML files (Table 9):

1. The architecture file `regen/architecture.toml`, one per repository. It holds the allowed imports between subpackages, the known import cycles, and the pace policy.
2. The unit records `regen/units/<unit>.toml`, one per piloted module. Each holds the unit’s public contract, pace, risk, dependencies, intent items, evaluations and consumers.
3. The decision records `regen/decisions/<id>.toml`, one per decision. Each holds the driver, the decision, the rejected alternatives, the approval, the origin, and a review date.

Example 2.3 shows an excerpt of the first kind. All of them are plain text in a small, fixed vocabulary, so any editor will do and every change goes through the same merge request as the code it describes. TOML was chosen for the reasons the Python packaging standards give for `pyproject.toml`: it is readable by people and less complex than YAML (Cannon et al. 2016). It maps unambiguously to a table (Preston-Werner et al. 2021).

These TOML files are processed by `spotregen`, whose subcommand `check` is the static check (Section 2.4). `spotregen` never changes an existing file: the only subcommand that writes TOML, `new-unit`, drafts a unit record for a module that has none and refuses one that is already recorded (Example 2.14).

Definition 2.25 (Draft). A draft is a TOML file that `spotregen` wrote from the source tree, without any human review. It is a scaffold for a unit record, which a person must complete and approve before the static check accepts it.

A draft is not a unit record. The static check rejects it with UNIT06 until it contains at least one intent item.

For reading, `spotregen` uses the standard library parser `tomllib`, which deliberately has no writer (Hukkinen and Jain 2022), so comments, key order, and line breaks that a person chose are never rewritten by a program.

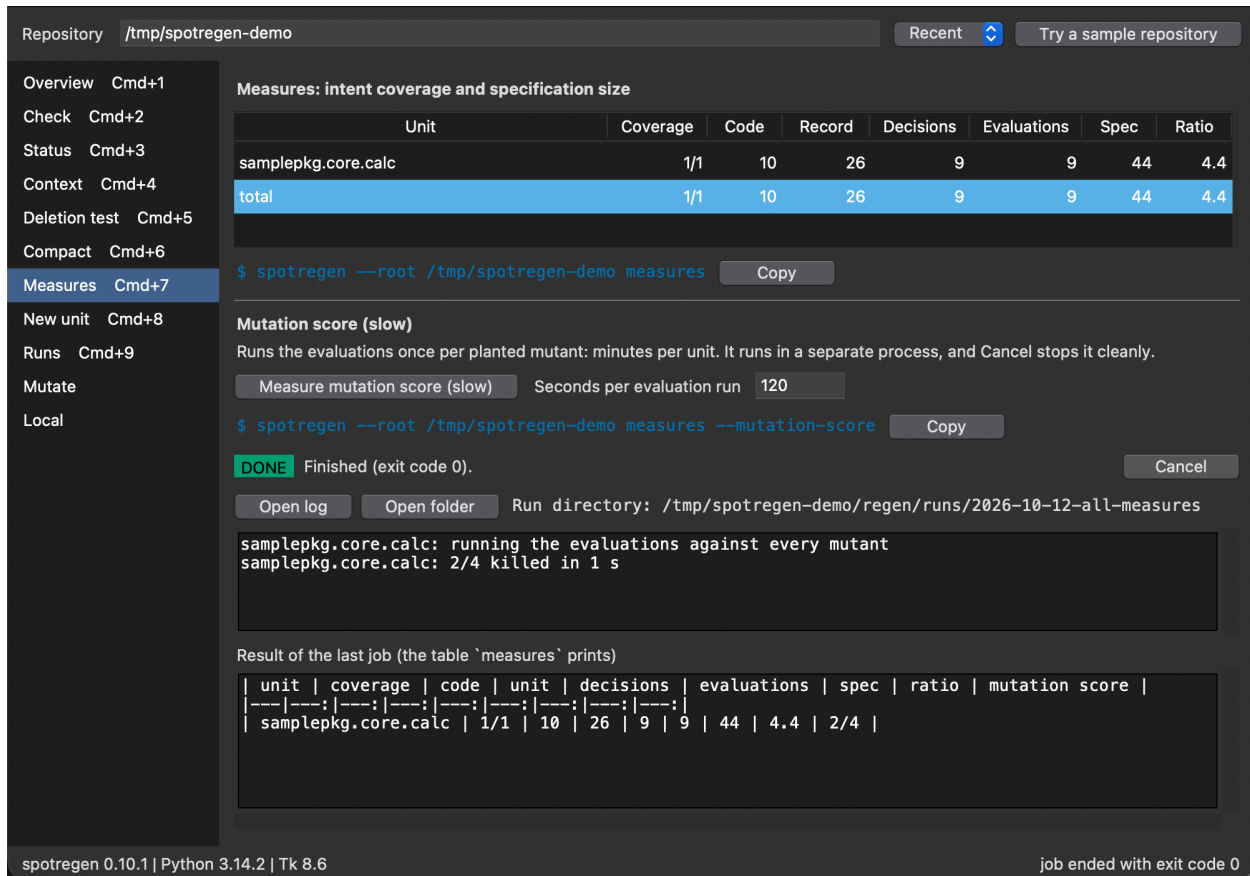


Figure 8: The Measures view after a mutation-score job on the sample repository. The log shows the progress lines. The table below is the output of `measures --mutation-score`.

Editing follows a short loop: change the file, run `make regen-check`, and read the findings. Mistakes are caught at two levels. A file that is not valid TOML, lacks a required field (`summary`, `reason`, `approved_by`, ...) or uses a value outside the vocabulary (`pace = "glacial"`) stops `spotregen` with exit code 2 and names the file and the field. A file that is well formed but disagrees with the tree yields the findings of Section 2.4, for example `DEL01` for an undeclared consumer or `EVAL04` for an intent item without an evaluation. One pitfall is specific to TOML: a top-level key written after the first `[[intent]]` header silently becomes a field of that intent item. The loader then reports the top-level key as missing or invalid, which names the right key but not the cause. All top-level keys therefore come first.

Example 2.14 (Drafting a unit record from the tree). A new unit record does not have to be written manually from a blank page. The subcommand `new-unit` reads the module and drafts `regen/units/<unit>.toml` with everything the tree can tell: the path, the first paragraph of the module docstring as `summary`, the public names (`__all__`, otherwise the public functions, classes, and upper-case constants), the package imports as `depends_on`, test files named after the module as `unit_tests`, and every file that imports or names the module as a consumer. Run on a sandbox helper for the blending helper, which is not yet a unit:

```
$ spotregen new-unit processing.blend --pace medium --risk medium
wrote regen/units/processing.blend.toml
ERROR UNIT06 spotforecast2_safe.processing.blend: declares no intent item (a unit without
intent cannot be regenerated)
next: write the intent items and evaluations, then run spotregen check
[exit code 0]
unit = "spotforecast2_safe.processing.blend"
path = "src/spotforecast2_safe/processing/blend.py"
summary = """
Post-hoc blend of a model forecast with an external prior -- pure computation.
```

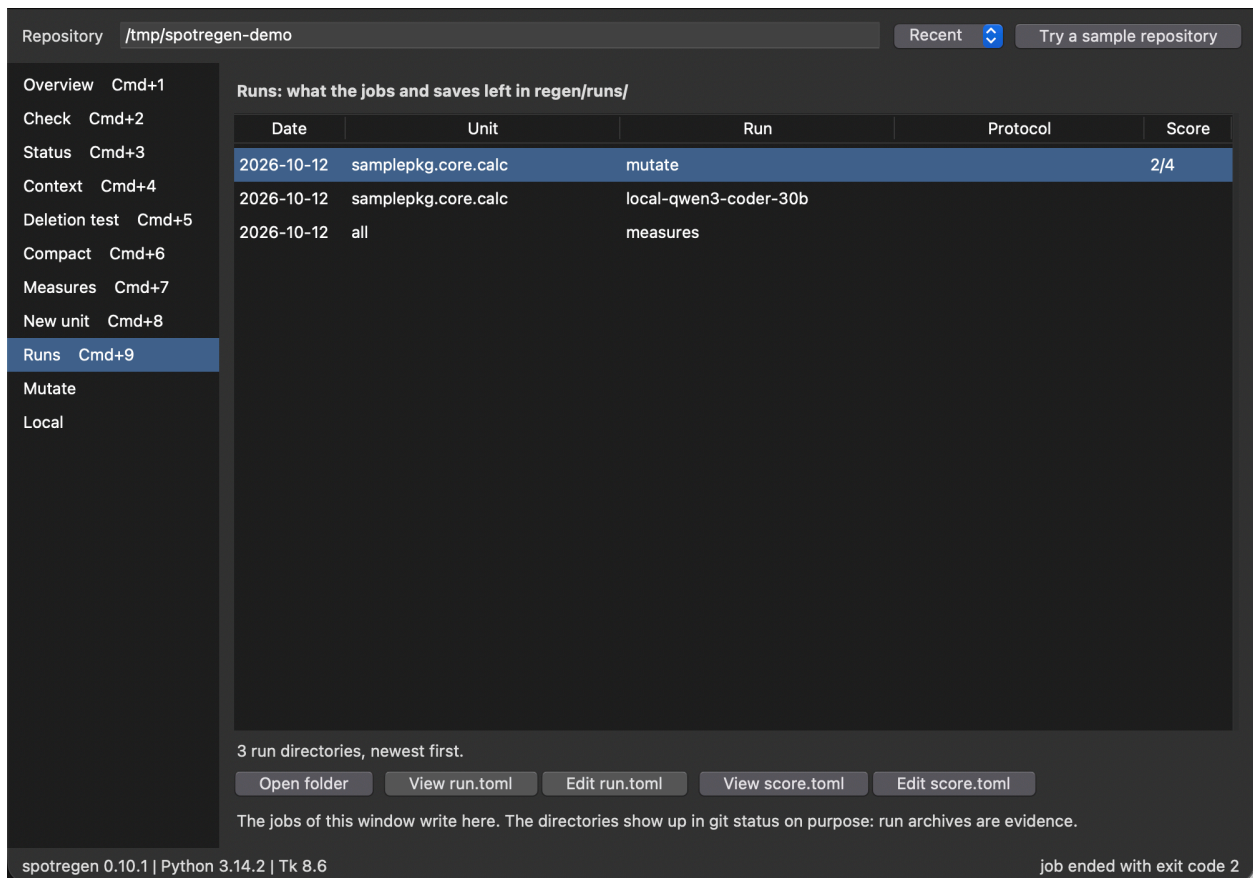


Figure 9: The Runs view on the sample repository after a Measures job, a Mutate job, and a Local job. The Local run stopped at once because no model server was running, so it recorded neither a protocol nor a score.

```
"""
pace = "medium"
risk = "medium"
public = ["blend_with_prior"]
depends_on = []
evaluations = [] # e.g. tests/test_regen_eval_blend.py, marked pytest.mark.regen
unit_tests = ["tests/processing/test_blend.py"]
```

The draft contains a commented template for the first intent item. The static check rejects the unit record with UNIT06 until one exists. A scaffold can therefore never pass the check unreviewed. What the subcommand `new-unit` cannot read from the tree is left to a person: the intent items, the evaluations, the decisions, and the external, operational, and human consumers.

Example 2.15 (Redrafting the pilot unit records). This experiment measures how much of a unit record `new-unit` can recover from the source tree alone. For each of the four pilot units, the hand-written record `regen/units/<unit>.toml` was deleted in a sandbox copy of the repository, `new-unit` drafted a new unit record with the same pace and risk, and the draft was compared field by field with the original. The module itself was left untouched. The original was restored before the next unit was drafted.

The experiment is not a deletion test. Drafting is not rebuilding. The deletion test of Fowler (2027, ch. 1) deletes the *implementation* and asks whether the system could be rebuilt correctly from the knowledge that remains. Here the *knowledge* is deleted and the implementation remains. The two directions also differ in how they work:

1. Drafting reads the code and writes a unit record. It is a deterministic static analysis of the module, its imports, and the files that name it. No model is involved, so the same tree always yields the same draft.

2. Rebuilding reads the records and writes code. A generator receives the regeneration bundle. The evaluations judge the result. Example 3 (Section 3.4) does this for the leakage guard.

The experiment answers the question the deletion test leaves open: which part of a unit record is a fact that the tree already holds, and which part is knowledge that would be lost with the record. Only the second part has to be written and reviewed by a person. The subcommand `deletion-test` (Definition 2.22) is again different: it deletes nothing and reports the blast radius and the knowledge held outside the implementation of one unit.

The columns of Table 16 are:

- `public`, whether the drafted public contract equals the original
- `depends_on`, whether the drafted package imports equal the original
- `found`, how many of the in-repository dependents of the original (consumers, unit tests, and evaluation files that the static check can see) the draft finds
- `external etc.`, the number of external, operational, and human consumers of the original, and
- `intent`, the number of its intent items. The last two columns count what the draft cannot contain.

Table 16: Drafts by `new-unit` compared with the hand-written unit records of the four pilot units. Found: in-repository dependents (consumers, unit tests, and evaluation files the static check can see). Added by hand: external, operational and human consumers, and intent items.

unit	public	depends_ on	found	ext., op., hum. consumers	intent
<code>multitask.guards</code>	same	same	5 of 5	18	8
<code>preprocessing.outlier</code>	same	same	9 of 9	10	8
<code>preprocessing.target_ corruption</code>	same	same	6 of 8	10	8
<code>processing.forecast_scoring</code>	same	same	6 of 6	7	8

The draft reproduces the public contract and the allowed imports of every pilot unit and finds every in-repository dependent that imports or names the module. The two dependents it misses are unit tests of the target-corruption detector that reach it only through the configuration classes and the multitask pipeline. They were therefore listed by hand.

Everything the draft gets right can be drafted again at any time, so losing it costs nothing. What the draft leaves empty would be lost with the unit record. A rebuild depends on it. The static check enforces it for the intent (UNIT06) but not for the dependents outside the repository, which it cannot see and can only take as declared.

The knowledge that makes a unit regenerable, its intent and its dependents outside the repository, is exactly the part the draft leaves open. This matches the division of labour that Fowler (2027, ch. 6) draws for decision records: a machine can collect facts, but a person has to supply the rationale.¹² Here the same holds for intent, what must stay true (Fowler 2027, ch. 3).

Decision records have no generator, on purpose. Their fields (driver, rejected alternatives, assumptions) are rationale, not facts that can be read from the tree. A new record is started by copying an existing one in `regen/decisions/`. Example 2.6 shows one of the decision records of the pilot. When an agent reconstructs a rationale from code and history, the record says so with `origin = "machine"`. In the pilot this applies to 10 of the 15 decision records (Section 2.9). A decision that a person takes when it is made is recorded with `origin = "human"` and names the person in `approved_by` (Example 2.16).

Example 2.16 (A decision record written by a person). The decision to move the regeneration tooling from scripts copied into each repository to the package `spotregen` was taken and recorded by the maintainer on the day it was made (long fields abbreviated):

¹²“Decision capture has to be a baked-in part of the workflow, or it won’t happen. [...] The one thing you can’t automate is the business rationale: the reason this tradeoff was acceptable. That is the part most worth a human’s time” (Fowler 2027, ch. 6).

²“A machine-captured fact (this prompt, this model, this evaluation result) is cheap to trust. A human-asserted rationale is a claim, and claims decay, mislead, or get written half-heartedly under deadline” (Fowler 2027, ch. 6).

```

title = "The regen tooling is the spotregen package, a dev-group dependency, not
↳ scripts/regen*.py"
date = 2026-10-01
status = "active"
driver = "scripts/regen.py, regen_blind.py and regen_local.py were maintained as separate
↳ copies in spotforecast2-safe and mac12l (mac12l carried a packaged ...)"
decision = "spotregen>=0.1.0 is in [dependency-groups] dev only, hash-pinned in uv.lock.
↳ make regen-check runs uv run spotregen check. scripts/regen.py, ..."
invariant = "spotregen stays outside the safety-critical wheel: it is not in
↳ [project].dependencies or optional dependencies, so the wheel contents and the SBOM
↳ ..."
approved_by = "Thomas Bartz-Beielstein, 2026-10-01"
origin = "human"
review_after = 2027-09-30
evidence = [
    "KB:2026-10-01-spotregen-package-adr",
    "file:pyproject.toml",
    "file:tests/test_regen_spec.py",
    "url:https://pypi.org/project/spotregen/0.1.0/",
]
assumes = [
    "spotregen stays standard-library only with no transitive dependencies",
    "spotregen stays out of the wheel and the SBOM (dev group only)",
    "a PyPI artefact in the gate is acceptable because uv.lock pins its hash",
]

[[alternatives]]
option = "keep scripts/regen*.py as a per-repository copy"
rejected_because = "the copies drifted between spotforecast2-safe and mac12l and were
↳ tested twice"

[[alternatives]]
option = "ship spotregen in the package (spotforecast2_safe.regen)"
rejected_because = "developer tooling would enter the safety-critical wheel, its SBOM and
↳ the MODEL_CARD scope"

[[alternatives]]
option = "spotregen as a runtime or optional dependency"
rejected_because = "it would change the wheel metadata and the SBOM; the gate needs it
↳ only in development"

```

Compared with the reconstructed record of Example 2.6, three fields differ in kind. `approved_by` names a person and a date instead of the sources an agent read, so `origin = "human"`. The `assumes` list states conditions the maintainer can check, for example that `spotregen` stays standard-library only. When one of them no longer holds, the decision has to be revisited before its review date. And all three rejected alternatives were considered at the time of the decision, whereas a reconstructed record can list only the alternatives that left a trace in code or history.

2.6 Evaluations as an oracle that outlives the code

Judging a rebuilt module requires an oracle that does not depend on the module. This is the test oracle problem (Barr et al. 2015). For most of software history the problem could be papered over, Fowler (2027, ch. 5) observes, “because code changed slowly and a human who understood it stayed close by”. Regeneration “removes both cushions”: slow change and the human nearby. When a component is deleted and rebuilt on demand, “possibly by a machine, possibly many times a year”, the definition of what “still works” cannot live inside it, “because the component is the thing being replaced”. It “has to live somewhere else, in a form that outlasts the code it judges”. That place is what Fowler calls the evaluations.

Following Fowler (2027, ch. 3), who separates tests that describe “the system’s promises” from tests that describe “the current implementation’s habits”, we separated two kinds of tests.

Definition 2.26 (Evaluation test). An *evaluation test*, or simply an evaluation, is the form the evaluation primitive (Definition 2.18) takes in this study: a durable evaluation (Definition 2.4) written as a pytest test in an evaluation file of a unit and marked `regen`. Three properties turn Fowler’s “executable claim about system behavior that should remain meaningful across implementations” into something that can be checked: the test imports only the names the unit declares as public (EVAL03), it states one promise of an intent item that names it (EVAL04, EVAL05), and it must survive a rewrite of the module in a different style (shown by the regenerations, Section 3.4).

The primitive is wider than its form here. Fowler (2027, ch. 5) also counts online evaluation, which “scores the system against live traffic as it happens”. Our evaluation tests are batch evaluations only, the kind that Fowler calls the gate a regenerated component must clear (Fowler 2027, ch. 5). They are the only judge of a rebuilt module (Figure 1).

Both kinds of tests are verification in the classic sense: they check an implementation against something written down beforehand (“are we building the product right?”). They differ in what they check against. Validation asks a different question (“are we building the right product?”) and applies to the intent itself (Table 17).

Table 17: Unit tests, evaluations, and the review of the intent items, by what they check and what they check it against.

	checks	against	kind
Unit test	the current implementation	its design: private helpers, internal details	verification against the design
Evaluation	any implementation	the intent items (the specification)	verification against the specification
Review of the intent items	the intent itself	the needs of operators, incidents, regulation	validation

Fowler (2027, ch. 4) uses the terms in the same way. He applies validation to the intent, not to the code:

Validation, then, asks three questions of the intent: is it complete, is it unambiguous, is it testable.

— Fowler (2027, ch. 4)

He places evaluations within verification (Fowler 2027, ch. 4): “Generation is safe because the result can be judged against durable evaluations; verification grades itself against them”. He also states the limit of evaluations, which is the gap that validation has to close:

Mutation testing can tell you your suite is weak, but it cannot tell you your suite is testing the wrong thing, because it only measures whether you would catch a change, not whether the behavior you locked in was ever the behavior you wanted.

— Fowler (2027, ch. 5)

No test answers this question. In this study, validation happens in three places: in the decision records, where a person supplies the reason; in the gap reports of the regenerations, which ask Fowler’s question whether the intent is complete; and in the specification-first change (Definition 2.30), where the maintainer decided that the leakage guard must fail closed. The rebuilds of Section 3.4 show why the distinction matters: the original and a rebuild can pass the same evaluations, yet behave differently on inputs that the intent does not cover. Whether that difference matters is a question of validation.

Both tests are kept, but only evaluations are part of the regeneration bundle. Only they decide whether a rebuilt module is accepted. Evaluations live in `tests/test_regen_eval_<unit>.py` and carry the pytest marker `regen`.

Definition 2.27 (Metamorphic relation). Where no labelled ground truth exists, an evaluation asserts a *metamorphic relation* (Segura et al. 2016): how the output must change, or not change, under a transformation of the input.

Fowler (2027, ch. 5) names metamorphic testing as the answer to the second hard question of evaluation design, “what you do when there is no oracle at all, when you genuinely cannot state the correct output in advance?” His answer: “Instead of asserting a specific answer, you assert a relationship that must hold between runs”.

You do not need to know the right answer to know that the right answer must obey those rules. When an exact oracle is out of reach, these relations are frequently the only oracle you can build, and they are often stronger than the example-based tests teams reach for by reflex, because a single relation constrains an entire space of inputs at once instead of pinning down one case.

— Fowler (2027, ch. 5)

Three of the four pilot units use such relations: the target-corruption detector (TC-2), forecast scoring (FS-2, FS-3, FS-5) and the outlier detector (OUT-3). The leakage guard needs none, because set membership is an exact oracle. *spotregen* treats a metamorphic evaluation like any other evaluation. The relation is stated in the intent item and in the test code. Where an incident is on record, the incident is replayed as a regression evaluation with the original timestamps and values.

2.7 Experimental protocols

Three protocols (Definition 2.5) were used. Two of them answer the three questions (QF-1, QF-2, QF-3) of the deletion test (Section 1) as follows:

1. *Regeneration* answers whether the knowledge is complete (QF-1), because the generator lists where the bundle was silent. It also answers whether a rebuild keeps the promises (QF-2), because the rebuild is judged by the evaluations.
2. *Mutation analysis* answers whether a replacement can be judged (QF-3): the evaluations must notice a wrong implementation.

The third protocol, *specification-first change*, requires that no behaviour changes occur without a change of the knowledge first. This protocol therefore serves as a check on the specification itself. It is not part of the deletion test.

2.7.1 The mutation analysis protocol

The first protocol is mutation analysis (DeMillo et al. 1978; Jia and Harman 2011). Coverage alone is not sufficient: it measures how much code ran, not whether an evaluation would notice a wrong result, as Fowler (2027, ch. 5) states:

The first question is: how you know your suite is good enough to gate a regeneration? Coverage will not tell you, because coverage measures how much code ran, not whether your evaluations would notice if it were wrong. Mutation testing tells you. It works by deliberately breaking the code in small ways, flipping a comparison, changing a constant, and running your suite against the sabotaged version. If the suite still passes, it has just proved it would not have caught that bug, and every surviving mutation is a hole in your oracle. Mutation testing is how you test the tests, and for a suite that has to be trusted with deletion and regeneration, it is not optional.

— Fowler (2027, ch. 5)

Fowler (2027, ch. 5) pairs mutation testing with metamorphic testing (Definition 2.27): mutation analysis judges the evaluations, whereas metamorphic relations make evaluations possible where no oracle exists.

Definition 2.28 (Mutation analysis). For each pilot unit, plausible bugs were planted in a scratch copy (a flipped comparison, a dropped branch, a mutated input) and the evaluations were required to kill every mutant. The mutants were planted by the LLM agents that wrote the specifications (Section 2.9). A second, independent sample was generated afterwards by *spotregen mutate*, which applies fixed mutation operators to every eligible node of the module.

2.7.2 The regeneration protocol

The second protocol is regeneration (Fowler 2027). It carries out the part of the deletion test that *spotregen deletion-test* only prepares: the implementation is actually deleted, rebuilt and judged (components 3 and 4 of Definition 2.22).

Definition 2.29 (Regeneration). *Regeneration* is implemented as a three-step experiment. The implementation of a unit is deleted in a separate git worktree. A generator (here, an LLM agent, Section 2.9) receives only the bundle written by *spotregen context*. The generator must be *memoryless*: it must not read the deleted file, its unit tests or the history. It must also list every place where the bundle was silent or ambiguous. The rebuild is judged first by the evaluations and then, as a second opinion, by the old unit tests. A failing unit test reveals either an implementation detail or a promise the specification never recorded.

Every generator of this study is memoryless. Only the Claude runs are called *blind*, because Claude Code can read the repository unless it is confined: in phase 1a the blindness was requested in the prompt, in phase 1b it was enforced by the harness (Section 3.4.1, Section 3.4.2). The local models of phases 3 and 4 are memoryless by construction, because they see only what the driver puts into the prompt.

2.7.3 The specification-first change protocol

The third protocol is the specification-first change.

Definition 2.30 (Specification-first change). A behaviour change starts with a decision record, then changes the intent items, then adds evaluations that must fail on the old code, and only then changes the code. A hand edit of the code that no intent item, contract, or evaluation explains is drift. Drift is what makes a module unregenerable. Fowler (2027, ch. 4) does not forbid such an edit but labels it: as a fix to be harvested back into the intent, a signed waiver or a patch that expires. Intent item TC-3 (Example 2.4) is such a harvested fix.

2.8 Measures of the pilot

The pilot is described by three measures of our own, namely intent coverage, mutation score, and specification size. Fowler (2027, ch. 3 and 5) motivates all three. The results report them for every pilot unit: intent coverage and mutation score in Table 24 (Section 3.1), and specification size in Table 23. Section 4.2 reads the three together.

spotregen computes the three measures with two subcommands:

- `spotregen measures` (functions `unit_measures` and `measures_table`) prints the intent coverage and the specification size of every unit. It only reads files and is deterministic. Intent coverage is also enforced elsewhere: `spotregen check` reports an intent item without an evaluation (EVAL04) and a named evaluation that does not exist (EVAL01), so a clean static check implies full coverage. `spotregen deletion-test` gives the verdict “regenerable” only if the coverage is complete.
- `spotregen mutate UNIT` (function `mutation_score`) reports the mutation score (Definition 2.32) of one unit on mutants it generates itself (details below). The option `measures --mutation-score` runs it for every unit and appends the score as a last column (Section 3.1).

Mutants can also be planted by hand. The pilot therefore has two samples of mutants (Definition 2.28): the ones planted by hand during the pilot and the ones generated by `spotregen mutate`.

2.8.1 The intent coverage measure

Definition 2.31 (Intent coverage). The *intent coverage* of a unit is the share of its intent items that name at least one existing durable evaluation. It measures the first half of Fowler’s warning that “Intent without evaluations is aspiration, and evaluations without intent are a pile of checks with no theory” (Fowler 2027, ch. 3).

Intent coverage is not code coverage: it counts the number of promises that have a judge, not lines that ran. The static check reports an intent item without an evaluation as error EVAL04 and a missing evaluation as error EVAL01, so a clean check implies full intent coverage. It also differs from the evaluation coverage of Fowler (2027, ch. 2), which asks whether any evaluation notices that a component has been deleted.

Example 2.17 (Intent coverage of the leakage guard).

The leakage guard `multitask.guards` declares 8 intent items, LG-1 to LG-8. Each names between 2 and 8 durable evaluations, and all of them exist, so its intent coverage is 8/8. If the evaluations of LG-8 were removed from the unit record, the static check would report EVAL04 and the coverage would drop to 7/8, even if the remaining evaluations still executed every line of the guard. Fowler’s evaluation coverage, by contrast, would stay complete: every evaluation imports the guard, so deleting the module fails all of them. A more telling probe replaces the module by a stub that “satisfies the shape of the contract while doing nothing real” (Fowler 2027, ch. 4), which every evaluation should reject.

Table 18 lists the intent items of the leakage guard.

Table 18: Intent items of the leakage guard `multitask.guards`. *Scope* is the field of the unit record, *invariant* a summary of its field *invariant*, and the last two columns count the evaluations and decision records the item names.

ID	scope	invariant	named evaluations	decisions
LG-1	<code>assert_no_leakage</code> , the three surfaces	Each of the three surfaces (training frame, selected exogenous features, fitted features of every target) is checked on its own. A forbidden name on any of them raises <code>LeakageError</code> .	5	1
LG-2	<code>assert_no_leakage</code> , the <code>LeakageError</code>	One <code>LeakageError</code> names the task, every leaking surface and its forbidden columns, in a fixed message frame.	4	2

ID	scope	invariant	named evaluations	decisions
LG-3	fitted-feature-name reading, per estimator	Fitted feature names are read from the first attribute present, covering LightGBM, CatBoost, scikit-learn and XGBoost.	3	1
LG-4	assert_no_leakage, unreadable fitted features	A surface that cannot be read raises a RuntimeError that is not a LeakageError and names the cause. The guard never passes silently.	8	3
LG-5	assert_no_leakage, the clean case	Without a forbidden name the guard returns None and has no other effect.	3	1
LG-6	matching of column names	Names match only by exact string equality, so derived names are no violation.	2	1
LG-7	assert_no_leakage, all calls	The guard is read-only and deterministic: the same inputs give the same outcome and a byte-identical message.	2	1
LG-8	assert_no_leakage, a leak next to an unreadable surface	A leak on a readable surface raises LeakageError even if another surface cannot be read.	3	1

2.8.2 The mutation score measure

Definition 2.32 (Mutation score). The *mutation score* of a unit is the share of the mutants of its module, planted by the mutation analysis protocol (Definition 2.28), that its durable evaluations kill. A mutant is killed if at least one evaluation fails or errors against it, or if the run exceeds its time limit. The measure differs from the mutation score of Jia and Harman (2011), the ratio of killed mutants to non-equivalent mutants, in two respects. Only the durable evaluations run, because they alone judge a rebuild. Equivalent mutants stay in the denominator, so the value is a lower bound. The score depends on the sample: Table 24 gives it for the hand-planted mutants and for the generated ones. It estimates the strength of the evaluations, whether they notice a wrong implementation (QF-3). It follows the advice of Fowler (2027, ch. 5) to judge a suite by mutation testing.

Mutation tests are performed as follows using `spotregen`: `spotregen mutate UNIT (function mutation_score)` measures the mutation score. It generates mutants of the unit’s module from its syntax tree (a flipped comparison, and for `or`, a dropped `not`, a negated condition, a changed constant or operator, a deleted call statement, `continue` or `break`, `raise` replaced by `pass`, a returned value replaced by `None`), writes each into a throw-away copy of the repository and runs the unit’s evaluations against it. It lists the surviving mutants, exits with 1 if there is one, and archives every result as TOML with `--out`. It runs the evaluations once per mutant, so it is slow and not part of the static check.

On the generated sample, the durable evaluations kill 282 of 345 mutants, an mutation score of 82 %. The review of the survivors found 15 of them equivalent. Without these, the score is 282/330, or 85 %.

Example 2.18 (Mutation score of the leakage guard).

For the leakage guard, 12 mutants were planted in the pilot and the durable evaluations killed 12 of them, a mutation score of 12/12. A mutant that adds a `break` after the first violation, so that the guard reports only the first violation it finds, fails a single evaluation, which is enough to kill it. Had that evaluation been missing, the mutant would have survived, and its bug would have lowered the score of the guard.

2.8.3 The specification size measure

Definition 2.33 (Specification size). The *specification size* of a unit is the number of lines of its unit record, of the decision records its intent items cite, and of its evaluation files. It is compared with the number of lines of the implementation, because Fowler (2027, ch. 5) observes that evaluation tends to become “the larger part of the work, larger than the generation it was built to check”. Counting lines is our operationalisation of that observation.

Example 2.19 (Specification size of the leakage guard).

The leakage guard has 239 lines of implementation. Its unit record has 232 lines, the 4 decision records its intent items cite have 120, and its evaluation file has 453. Its specification size is therefore 805 lines, 3.4 times the implementation (`spotregen` measures).

2.9 Role of the language model during the five phases

Language models act in two kinds of steps. In the work on the specification (the pilot records, the planted mutants, and the specification-first change), the model was always Claude Opus 5.5 (`claude-opus-5-5`) in Claude Code, either in the maintainer’s session or as a subagent that the session spawned for one task. In the regeneration of Example 3 (Section 3.4), the generator and its harness changed four times in three days, each time for a reason the previous phase supplied. Table 19 introduces the five phases. Phases 1a and 1b used Claude. Phase 2 ran a local model in the Claude Code harness of phase 1b. Phases 3 and 4 ran local models with a driver of their own and without any Claude component (Section 2.9.1). The results of the phases and the reasons for each change follow in Table 25.

Table 19: The five phases of the regeneration in Example 3. The names are those of the subsections of Section 3.4.

phase	date	generator	harness
1a: Claude, blind by request (Section 3.4.1)	2026-09-29	Claude Opus 5.5	subagent of a Claude Code session, blindness requested in the prompt
1b: Claude with enforced blindness (Section 3.4.2)	2026-09-30	Claude Opus 5.5	<code>claude -p</code> with <code>--restricted</code> and <code>--safe-mode</code>
2: a local model in the Claude Code harness (Section 3.4.3)	2026-09-30	Qwen3.8-27B, reasoning on	the harness of phase 1b, pointed at a local endpoint
3: local models in short sessions (Section 3.4.4)	2026-09-30	Qwen3.8-27B (reasoning off), Qwen3-Coder-30B	short sessions of an own driver, no Claude Code
4: Devstral against Qwen (Section 3.4.5)	2026-10-01	Devstral Small 2, Qwen3.8-27B	the driver of phase 3

Table 20 lists each step, what the model did there, and what judged the result. The split follows one rule. The LLM writes knowledge and code, but nothing it writes is accepted on its own authority. Every artefact passes a deterministic check (the static check, the durable evaluations, and the old unit tests as a second opinion). Every safety decision is taken by the maintainer.

Definition 2.34 (Canary test). A *canary test* checks that a generator session is confined to the context it is given. It is a session of its own, run before the regeneration runs, whose prompt asks the model to attempt seven actions once each: read the original module, list the parent directory, read a settings file, run a shell command, and write outside the workspace, which must fail, and read and write inside the workspace, which must succeed. The test passes on the records of the harness or driver (refused tool calls, created files, remote addresses), never on the model’s own account.

Table 20: Where an LLM acts in the study and what checks its output. Unless a row names another model, the LLM is Claude Opus 5.5.

step	what the LLM did	what judged the result
Drafting a unit record (<code>new-unit</code>)	nothing: the draft is read from the source tree	UNIT06, the maintainer
Intent items, evaluations, and decision records of the pilot	one subagent per unit reconstructed them from code, unit tests, KB, and history	the static check, the maintainer’s review, <code>origin = "machine"</code> , and review dates
Mutation analysis (Definition 2.28)	the same subagents planted the mutants	the evaluations
Phase 1a: Claude, blind by request (Section 3.4.1)	a subagent rebuilt the module from the bundle and listed the gaps	the evaluations, the old unit tests (second opinion), the tool-call log
Phase 1b: Claude with enforced blindness (Section 3.4.2)	<code>claude -p</code> , confined by the harness, rebuilt the module three times	a canary test of the confinement, the evaluations, the old unit tests (second opinion)
Phase 2: a local model in the Claude Code harness (Section 3.4.3)	nothing: Qwen3.8-27B in the same harness wrote no module in two attempts	the canary test

step	what the LLM did	what judged the result
Phase 3: local models in short sessions (Section 3.4.4)	Qwen3.8-27B and Qwen3-Coder-30B, driven by <code>regen_local.py</code> without tools, ran three regenerations each	the canary test, the socket log, the evaluations, the old unit tests (second opinion)
Phase 4: Devstral against Qwen (Section 3.4.5)	Devstral Small 2 and Qwen3.8-27B ran three regenerations each under a decision rule set before the runs (Definition 3.9)	the canary test, the socket log, the evaluations, the old unit tests (second opinion)
Safety decisions (Definition 2.16)	nothing: the generator raised them as questions	the maintainer (<code>origin = "human"</code>)
Specification-first change (Definition 2.30)	wrote the intent items, evaluations, and code after the decision	new evaluations that had to fail on the old code, the static check
Static check, deletion test, bundle, compaction	nothing: standard-library code	–
This report and its German version	drafting, research, and translation	the author

2.9.1 A local architecture with short sessions

The development of the `spotreg` package can be divided into four phases. Phases 3 and 4 used a generator architecture of their own. In phase 2 (Section 3.4.3) a local model in the Claude Code harness wrote no module. The reasoning mode and a harness built for Claude were suspected. The architecture therefore gives the local model a harness of its own, with the reasoning switched off and without any Claude component: no Claude Code, no Agent SDK, no Anthropic application programming interface (API). It was built for phase 3 as the script `regen_local.py` of `spotforecast2-safe`. Phase 4 added two options to it. On 2026-10-01 the scripts of all phases (`regen.py`, `regen_blind.py`, `regen_local.py`) were replaced by the `spotregen` commands `context`, `claude` and `local`. `spotregen local` keeps the prompts and defaults of `regen_local.py` and adds only a run of the dependents (Section 3.4.7). The architecture rests on three design decisions.

Design decision 1: a deterministic driver instead of an agent session. Each model step is a fresh request that receives only the context of that step. Its result is a file. The next step reads that file, not a chat history (Figure 10). The steps without a model are those of `spotregen`: `spotregen context` builds the bundle. The evaluations run in a detached git worktree in which the module is deleted.

Design decision 2: confinement by construction. The model is offered no tools. It sees only the text the driver puts into its prompt, the bundle, and the files named with `--allow`, so it cannot read the deleted module, however it is prompted. The canary test therefore reduces to two checks: the canary prompt must produce no file and every network connection of the driver must go to the local server. The driver refuses any endpoint that is not on localhost and records every remote address of its sockets, sampled with `lsof`, in `run.toml`. The gap report, which the Claude generator wrote at the end of its single session, becomes a session of its own.

Design decision 3: a harness that fits the model. The model server LM Studio serves the model on the maintainer’s laptop. The driver talks to its OpenAI-compatible completion endpoint with the standard library and renders the chat prompt itself, which makes the reasoning mode controllable. Qwen3.8-27B reasons at the highest effort by default. Neither the request parameters nor LM Studio’s own API switched this off, so the driver writes the empty reasoning block, which the chat template renders for `enable_thinking=false`, into the prompt. Sampling is the same for every model: temperature 0.7, top-p 0.8 and top-k 20, the values of the generation configuration of Qwen3-Coder, with the run number as seed.

3 Results

All examples in Section 3 are executable cells of this document. Examples that need a broken repository (a planted import, a planted bug, an expired decision) run on a throw-away copy of the relevant directories created in a temporary directory, so the working tree is never modified. The report lives in its own repository, `bart26x-regen`. It reads a checkout of `spotforecast2-safe`. Its helper module `bart26x/_regen_demo.py` contains the sandbox and the commands, which run with the interpreter of that checkout.

Redoing the examples. A reader needs git, uv, and Python 3.13 or later. The reader clones `spotforecast2-safe` at the commit this document reads. The clone is the working directory of every “Running the example” paragraph below. The shell function `edit` replaces one exact string in a file and fails if the string does not occur exactly once. `git checkout -- regen` undoes an edit:

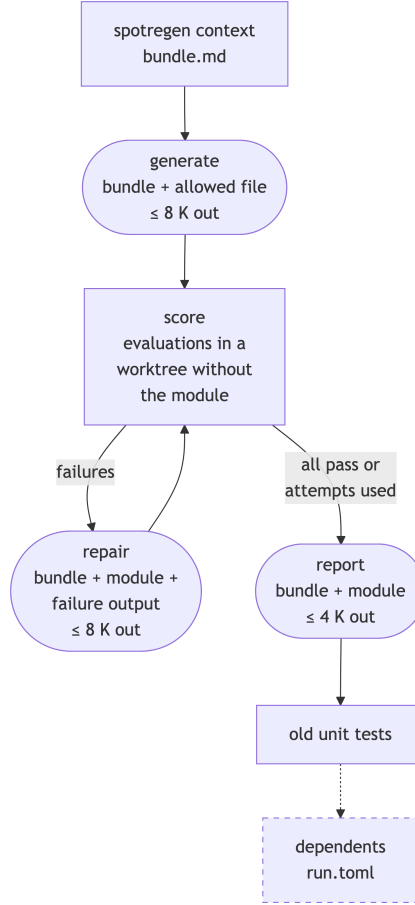


Figure 10: Steps of `spotregen local` with its default budgets. Rounded boxes are fresh model sessions without history, each limited to 15 minutes. Rectangles run without a model. Only files cross from one step to the next. Every model session also receives the prompt. A repair receives the last 12 000 characters of the failure output. Phases 3 and 4 ran these steps with the predecessor `regen_local.py`, without the dependents (dashed).

```

git clone https://git-ce.rwth-aachen.de/thomas.bartz-beielstein/spotforecast2-safe.git
cd spotforecast2-safe
git checkout 41c0e38
uv sync --group dev      # installs spotregen 0.10.1
uv run spotregen check   # the static check: 0 errors
edit() { uv run python -c "import pathlib, sys; p = pathlib.Path(sys.argv[1]); \
    t = p.read_text(); assert t.count(sys.argv[2]) == 1; \
    p.write_text(t.replace(sys.argv[2], sys.argv[3]))" "$@"; }

```

The static check resolves the KB: evidence of a decision record only where the knowledge base is present (`kb_dir` in `regen/architecture.toml`). On a reader's machine it is absent, so `spotregen check` passes without checking that evidence. The file and commit evidence is checked everywhere.

3.1 Coverage, evaluation, and specification size of the pilot

Regarding the three measures of Section 2.8, intent coverage, mutation score, and specification size, we present the following results. The following terms will be used in the tables and text: *Unit* names the module by a short name. The caption of Table 24 gives the full names. *Pace* and *risk* are the classes the unit record assigns (Section 2.3.5). The following definitions will be used:

Definition 3.1 (Intent items). *Intent items* counts the intent items of the unit record.

Definition 3.2 (Decisions). *Decisions* counts the decision records that the intent items of the unit cite, each counted once.

Definition 3.3 (Consumers). *Consumers* counts the consumers that the unit record declares, of every kind (Section 2.3.6).

Definition 3.4 (Evaluations). *Evaluations* counts the durable evaluations of the unit as pytest collects them, so each parametrised case counts separately.

The next four columns give the three measures.

Definition 3.5 (Intent coverage (column)). *Intent coverage* is the number of intent items that name an existing evaluation, out of all intent items (Definition 2.31).

Definition 3.6 (Score, by hand). *Score, by hand* gives the mutation score (Definition 2.32) as killed out of planted mutants, for the mutants planted by hand in the pilot (Definition 2.28).

Definition 3.7 (Score, generated). *Score, generated* gives the mutation score (Definition 2.32) as killed out of planted mutants, for the mutants generated by `spotregen mutate` (Definition 2.28).

Definition 3.8 (Specification size (column)). *Specification size* gives the lines of the unit record, the cited decision records, and the evaluation files, followed in parentheses by their ratio to the lines of the implementation (Definition 2.33).

All values except the hand-planted mutants, which were not archived, are read from the repository while this document renders.

3.1.1 Intent coverage

Every intent item is bound to at least one evaluation, so the intent coverage (Definition 2.31) of every unit is complete (32/32 in total). This is the precondition for a positive deletion-test verdict.

Table 21 gives the intent coverage of each pilot unit.

Table 21: Intent coverage per pilot unit (Definition 3.5). *Intent items* is as in Table 24 (Definition 3.1). *Covered* counts the intent items that name at least one existing evaluation, *named evaluations* the distinct evaluations the intent items name. Coverage is taken from `spotregen measures`.

unit	intent items	covered	named evaluations	intent coverage
<code>multitask.guards</code>	8	8	30	8/8
<code>preprocessing.outlier</code>	8	8	22	8/8
<code>preprocessing.target_corruption</code>	8	8	33	8/8
<code>processing.forecast_scoring</code>	8	8	23	8/8
total	32	32	108	32/32

3.1.2 Mutation score

The durable evaluations kill all 42 mutants planted by hand, but only 282 of the 345 generated by `spotregen mutate` (82 %). On the generated sample, only the leakage guard keeps a mutation score of 100 % (Definition 2.32).

Table 22 gives the mutation score of each pilot unit for both samples of mutants.

Table 22: Mutation score per pilot unit (Definition 2.32), as killed out of planted mutants and in percent. *Score, by hand* is for the mutants planted by hand in the pilot (Definition 3.6), which were not archived, *score, generated* for those of the archived `spotregen mutate` runs in `regen/runs/2026-10-02-mutation` (Definition 3.7).

unit	score, by hand	score, generated
<code>multitask.guards</code>	12/12 (100 %)	26/26 (100 %)
<code>preprocessing.outlier</code>	6/6 (100 %)	38/61 (62 %)
<code>preprocessing.target_corruption</code>	14/14 (100 %)	107/135 (79 %)

unit	score, by hand	score, generated
<code>processing.forecast_scoring</code>	10/10 (100 %)	111/123 (90 %)
total	42/42 (100 %)	282/345 (82 %)

3.1.3 Specification size

The specification (Definition 2.33) of every unit is larger than its implementation, by a factor of 1.6 to 3.4. The evaluation files alone are longer than the code in every unit, which is in line with Fowler’s observation that evaluation tends to become the larger part of the work (Fowler 2027, ch. 5).

Table 23 gives the specification size (Definition 2.33) of each pilot unit next to the size of its implementation.

Table 23: Lines of implementation and of specification per pilot unit, as `spotregen` measures prints them. All columns except *ratio* count lines. *Specification size* is the sum of the lines of the unit record, of the decision records its intent items cite (each counted once) and of the evaluation files. *Ratio* is specification lines per line of implementation.

unit	implementation	unit record	decision records	evaluation files	specification size	ratio
<code>multitask.guards</code>	239	232	120	453	805	3.4
<code>preprocessing.outlier</code>	354	189	83	395	667	1.9
<code>preprocessing.target_corruption</code>	624	203	92	716	1011	1.6
<code>processing.forecast_scoring</code>	517	167	79	597	843	1.6
total	1734	791	374	2161	3326	1.9

The knowledge held outside the implementation is about 1.9 times as large as the implementation itself, and most of it is evaluation code. The package `spotregen` 0.10.1 adds 11,688 lines of standard-library Python, shared by all units and repositories.

3.1.4 Summarizing the pilot results

Table 24 summarises the four pilot units as the repository defines them today.

Table 24: Pilot units and the three measures of Section 2.8. Units: `guards` = `multitask.guards`, `outlier` = `preprocessing.outlier`, `corruption` = `preprocessing.target_corruption`, `scoring` = `processing.forecast_scoring`, each in the package `spotforecast2_safe`. The columns are explained in the text.

unit	pace	risk	intent items	decisions	consumers	intent cover- age	score, by hand	score, gener- ated	specification size
<code>guards</code>	slow	high	8	4	22	44	8/8	12/12	805 (3.4)
<code>outlier</code>	medium	high	8	3	19	42	8/8	6/6	667 (1.9)
<code>corruption</code>	slow	high	8	3	16	40	8/8	14/14	1011 (1.6)
<code>scoring</code>	slow	high	8	3	10	51	8/8	10/10	843 (1.6)
total			32	13	67	177	32/32	42/42	282/345

The pilot units hold 32 intent items. Of the 15 decision records, 13 justify intent items of the pilot units, and 2 concern the layout of the knowledge itself. Of the 67 declared consumers, 37 are files in downstream repositories, 4 are operational processes such as the daily submissions, and 4 are people who read the unit’s messages. Almost half of the blast radius is therefore invisible from inside the repository.

Figure 11 shows the outcome of every mutant. Figure 12 shows the composition of the specification of each unit.

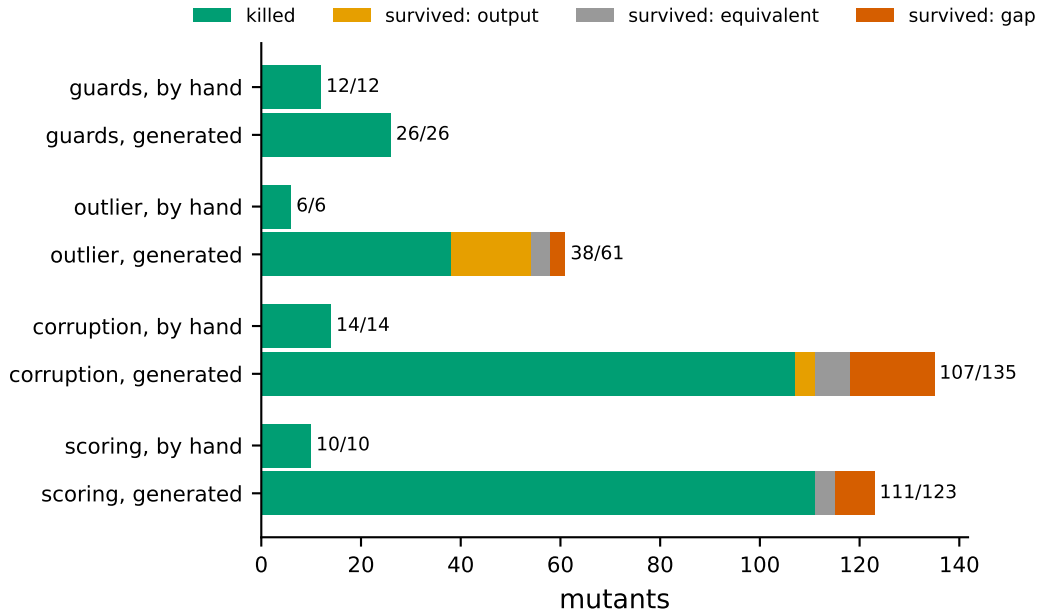


Figure 11: Outcome of the mutants of each pilot unit. Each unit has one bar for the mutants planted by hand and one for those generated by `spotregen mutate`. Survivors of the generated sample are split by the verdict of their review in `review.toml`: *output* changes only logs or prints, *equivalent* changes nothing observable, *gap* is behaviour that no evaluation pinned.

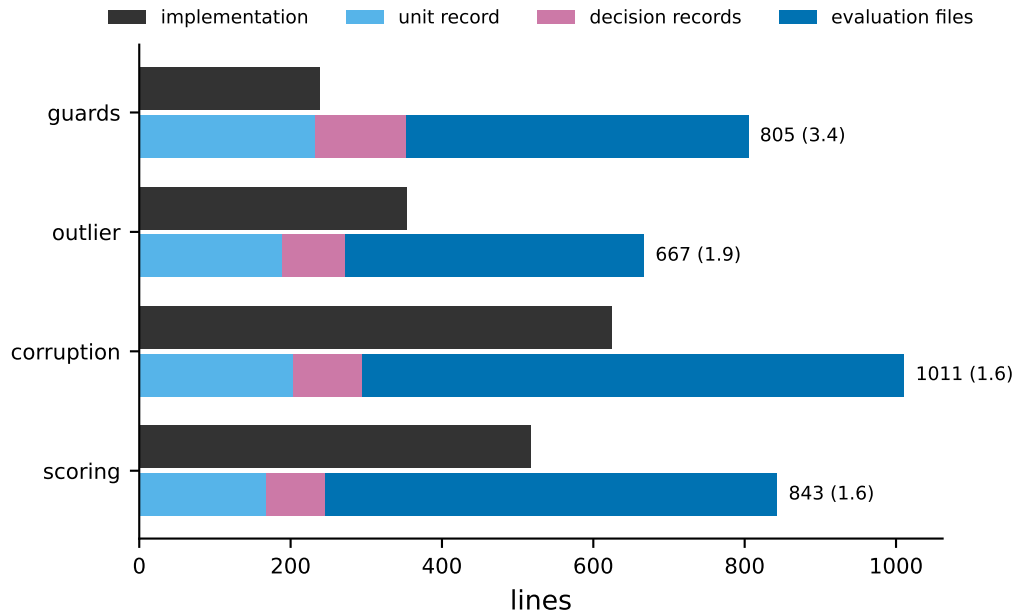


Figure 12: Specification size of each pilot unit (Definition 2.33), split into its three parts, next to the lines of the implementation. The number at the end of a bar is the ratio.

On the current tree the static check is clean and the durable evaluations pass:

```
$ spotregen check
```

```
regen check: 4 units, 0 errors, 0 warnings
[exit code 0]
```

The subcommand `spotregen status` prints pace, risk, intent coverage, decisions and consumers of every unit as a table, shown here as it prints it.

```
$ spotregen status
[exit code 0]
```

unit	pace	risk	intent	evaluated	decisions	consumers
spotforecast2_safe.multitask.guards	slow	high	8	8/8	4	22
spotforecast2_safe.preprocessing.target_corruption	slow	high	8	8/8	3	16
spotforecast2_safe.processing.forecast_scoring	slow	high	8	8/8	3	10
spotforecast2_safe.preprocessing.outlier	medium	high	8	8/8	3	19

The subcommand `spotregen measures` prints two of the three measures for every unit. Its columns are the intent coverage, the lines of the implementation (code), of the unit record (unit), of the cited decision records, and of the evaluation files, their sum (spec) and its ratio to code.

```
$ spotregen measures
[exit code 0]
```

unit	coverage	code	unit	decisions	evaluations	spec	ratio
spotforecast2_safe.multitask.guards	8/8	239	232	120	453	805	3.4
spotforecast2_safe.preprocessing.outlier	8/8	354	189	83	395	667	1.9
spotforecast2_safe.preprocessing.target_corruption	8/8	624	203	92	716	1011	1.6
spotforecast2_safe.processing.forecast_scoring	8/8	517	167	79	597	843	1.6
total	32/32	1734	791	374	2161	3326	1.9

The mutation score was kept out of `spotregen measures` by default. `measures` only reads files and finishes in a fraction of a second, whereas `spotregen mutate` runs the evaluations once per mutant. For the pilot units this took between 33 s and 4.4 min per unit, about 11 min in total in the archived runs. Table 24 therefore combines measures with the archived `mutate` runs. The option `measures --mutation-score` runs the mutation analysis of every unit and appends the mutation score as a last column. It is off by default for the same reason, so that `measures` stays a quick report that reads files. This document does not use it.

3.2 Example 1, compilation: one import, sixty-five findings

`processing` is a leaf of the import graph: `[allowed]` lists no dependencies for it (Figure 13). It is imported by `data` and `multitask` and imports nothing inside the package.

How the record came about. `[allowed]` and `known_cycles` in `regen/architecture.toml` are a baseline, not a design target. They were recorded on 2026-09-29 from the import graph of the tree at commit 6638156, as the comment above `[allowed]` says, in the commit that set up the specification (d96f6a1, which names Claude Opus 5.5 in Claude Code as co-author, Section 2.9). They describe what the package imported then, including 16 cycles that already existed as debt. The decision to keep this knowledge in `regen/` was the maintainer's (`regen-knowledge-layout`, `origin = "human"`). No language model takes part in the example itself, because the planted import was chosen by the authors and the static check is deterministic.

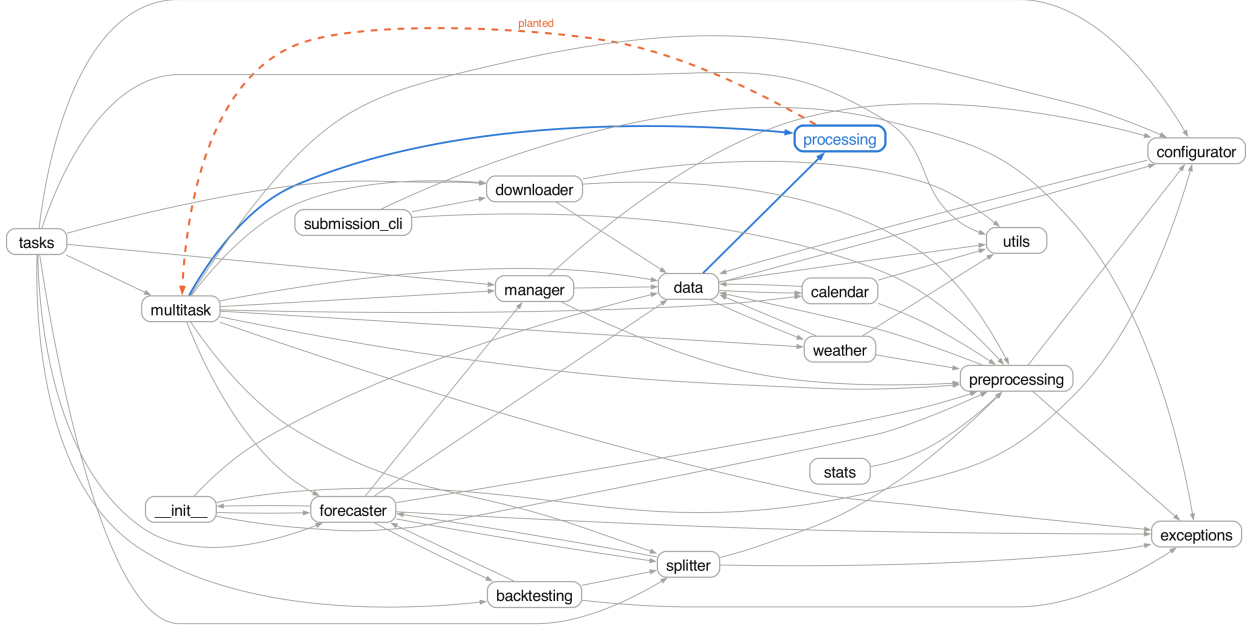


Figure 13: Allowed imports between the subpackages of `spotforecast2-safe`, as recorded in `[allowed]` of `regen/architecture.toml`. An arrow points from the importing to the imported subpackage. `processing` (blue) has incoming edges only. The dashed edge is the import planted below. Together with the edge from `multitask` it closes a cycle.

The following cell plants a single import of the `multitask` package into the scoring module of a sandbox copy and runs the static check.

```
with sandbox() as root:
    p = root / "src/spotforecast2_safe/processing/forecast_scoring.py"
    text = p.read_text()
    i = text.index("\nimport ")
    p.write_text(text[:i] + "\nfrom spotforecast2_safe.multitask import guards\n" +
    ↪ text[i:])
    code, out = regen("check", root=root)
    print(wrap("\n".join(ln for ln in out.splitlines() if "ARCH04" not in ln)))
    print("findings by check ID:", summarize(out))
```

```
ERROR ARCH01 src/spotforecast2_safe/processing/forecast_scoring.py: 'processing' imports
'spotforecast2_safe.multitask', but architecture.toml allows only []
ERROR DEL01 spotforecast2_safe.multitask.guards: src/spotforecast2_safe/processing/forec
ast_scoring.py depends on the unit but is not a declared consumer
ERROR UNIT04 spotforecast2_safe.processing.forecast_scoring: imports 'spotforecast2_safe.
multitask' which is not in depends_on (Compilation)
regen check: 4 units, 65 errors, 0 warnings
findings by check ID: {'ARCH01': 1, 'ARCH04': 62, 'DEL01': 1, 'UNIT04': 1}
```

The planted edge violates the architecture (ARCH01) and the unit's own `depends_on` (UNIT04), and it makes the scoring module an undeclared consumer of the leakage guard (DEL01). The dominant effect is ARCH04. Because `multitask` imports almost every other subpackage, the new edge closes 62 new import cycles at once.

A reviewer who only sees a one-line diff learns from the static check that the leaf is no longer a leaf. This is the architectural position that Fowler (2027, ch. 3) identifies as the real risk of a small change.

How the static check finds it. `spotregen` check does not import or run the package. It parses every module under `src/` with Python's `ast` module, maps each import to a subpackage and compares the edges with `[allowed]` (ARCH01) and with the `depends_on` of the units (UNIT04). For ARCH04 it computes which subpackages can reach

each other, directly or through others. Each such pair that is not in `known_cycles` is reported once as a new cycle. DEL01 comes from the consumer scan, which parses every Python file under `scan_dirs` (`src`, `tests`, `scripts`, `benchmarks`) in the same static way.

Running the example. The cell above works on a throw-away copy of the repository, so the report never changes the checkout it reads. A reader plants the import into the clone of Section 3 instead, runs the check and removes the import again:

```
git checkout 41c0e38
uv run python -c '
from pathlib import Path
p = Path("src/spotforecast2_safe/processing/forecast_scoring.py")
t = p.read_text(); i = t.index("\nimport ")
p.write_text(t[:i] + "\nfrom spotforecast2_safe.multitask import guards\n" + t[i:])
'
uv run spotregen check # exit code 1: 65 errors
git checkout -- src # remove the planted import
```

3.3 Example 2, the regeneration bundle

The bundle is the complete input a generator receives. It holds the architecture, the intent, the decisions, the public contract, the evaluations, and the consumers. It holds no implementation.

```
$ spotregen context multitask.guards -o <tmp>/bundle.md
wrote <tmp>/bundle.md
[exit code 0]
```

763 lines, 47,211 characters; sections:

```
Purpose
Architecture (compilation target)
Intent
Provenance
Public contract (signatures and docstrings, no bodies)
Durable evaluations
Consumers (must keep working)
```

```
code lines of the function bodies (25 characters or more): 45, in the bundle: 0
signature 'def assert_no_leakage' present in bundle: True
```

The last two lines are the essential property. The signature and the docstring of the public function are in the bundle, but no line of any function body is. A generator that passes the evaluations with this input has rebuilt the module from knowledge, not by copying it.

How the bundle is built. `spotregen context` reads the files of `regen/` and the module. It runs nothing. It writes the architecture (allowed imports, conventions, open choices, pace policy), the unit record with its intent items, the decision records the items cite, the public contract, the full source of the evaluation files and the list of consumers. The public contract is made with Python's `ast` module: only the names in the unit's `public` field are kept. Every function body is replaced by its docstring and . . . Private helpers are left out entirely. The output is deterministic, so the same checkout always gives the same bytes. This one has the SHA-256 `de4e12c0173cb571...` (commit `41c0e38`). No language model takes part in building the bundle. It is the input of the generator in Section 3.4.

What the bundle looks like. The excerpts below are cut from the bundle that the first cell of this example has just written. They follow one intent item, LG-5 (the clean case), from the intent through the public contract to the evaluation that checks it. They also show the beginning of each other section. [...] marks omitted lines. Long lines are wrapped.

```
# Regeneration bundle: spotforecast2_safe.multitask.guards
```

```
Write `src/spotforecast2_safe/multitask/guards.py` from scratch. You do not get the current
    implementation. Keep every public name and signature of the contract below, satisfy
    every intent item, and import nothing outside `depends_on`.
```

```
## Purpose
```

Data-governance leakage guard for the multi-target forecasting pipeline.
[...]

Architecture (compilation target)

- `multitask` may import from: calendar, configurator, data, downloader, exceptions, forecaster, manager, preprocessing, processing, splitter, weather
- this unit may import: spotforecast2_safe.exceptions
- open choice: internal helper decomposition and private names

[...]

Intent

LG-5

- scope: assert_no_leakage, the clean case
- invariant: When no forbidden name is on any surface the function returns None and has no other effect. An empty forbidden set never raises LeakageError. forbidden may be a set or a frozenset of str.
- reason: The submitters call the guard unconditionally after training and log 'leakage guard passed' on a normal return.
- checked by: test_clean_pipeline_passes_and_returns_none, test_empty_forbidden_set_never_raises_leakage, test_frozenset_forbidden_is_accepted

[...]

leakage-guard-fail-closed: The leakage guard fails closed: a readable leak always raises LeakageError, and missing surfaces or an empty target list are unreadable, not empty

- decision: All three surfaces are read before anything is raised. A forbidden name on any readable surface raises LeakageError (LG-8). Otherwise, an unreadable surface raises RuntimeError that is not a LeakageError (LG-4): fitted features that cannot be read, an empty mt.run_state.targets, and mt.data_with_exog or mt.exog_feature_names that is absent or None. An empty exog list stays a legitimate, readable surface.
- rejected: keep the lenient behaviour and document it (a leak in the training frame could reach a submission with only a warning when an estimator happened to be unreadable)

[...]

Public contract (signatures and docstrings, no bodies)

```
def assert_no_leakage(mt: object, forbidden: set[str], *, task: str) -> None:
    """Raise `LeakageError` if any forbidden column reached the model.
```

[...]

Durable evaluations

```
def test_clean_pipeline_passes_and_returns_none():
    mt = _mt()
    assert assert_no_leakage(mt, FORBIDDEN, task="spotoptim") is None
```

[...]

Consumers (must keep working)

- `src/spotforecast2\_safe/multitask/\_\_init\_\_.py` (code): re-exports assert_no_leakage as spotforecast2_safe.multitask.assert_no_leakage (in __all__)

[...]

The bundles of the archived runs. Each archived regeneration run keeps the bundle it gave the generator and records its SHA-256 as bundle_sha256. The runs of Section 3.4 were made before spotregen existed, with the script scripts/regen.py of spotforecast2-safe. Today's spotregen cannot read their unit records any more, because a field of the unit records has since been renamed. A bundle of those runs is therefore rebuilt at the commit the run names as its base, with the script of that commit. We did so for the comparison of 2026-10-01 (Section 3.4.5), whose

base is commit b152bde: the rebuilt bundle has the recorded hash, byte for byte. According to its run summary (summary.toml), the runs of 2026-09-30 received the same bundle.

Running the example. In the clone of Section 3:

```
uv run spotregen context multitask.guards -o bundle.md
shasum -a 256 bundle.md      # de4e12c0173cb571... at commit 41c0e38
grep -c 'unreadable.append(' bundle.md    # 0: a line of the body
git checkout b152bde && uv sync --group dev
uv run python scripts/regen.py context multitask.guards -o old.md
shasum -a 256 old.md         # 89cc60d517e92998...: the recorded bundle_sha256
git checkout 41c0e38 && uv sync --group dev
rm bundle.md old.md
```

3.4 Example 3, regenerating the leakage guard

This is the only example in which a language model acts. The module multitask/guards.py is deleted, a generator receives the bundle of Section 3.3 and one further file, and the durable evaluations judge what it writes.

The further file is src/spotforecast2_safe/exceptions.py, where spotforecast2-safe defines its own warning and exception classes (19 classes in 890 lines) and no line of the guard. Among them is LeakageError, which the guard raises when a forbidden column reaches the model. The bundle names the class in the public contract but does not contain it. A rebuild has to import it rather than define its own, because the submitters recognise a leak by exactly this class. Since LeakageError subclasses RuntimeError, the intent items speak of “a RuntimeError that is not a LeakageError” for an unreadable surface.

The generator and its harness changed four times in three days (Table 19). Table 25 lists the result of each phase and the reason for the next one.

Table 25: Results of the five phases of Table 19. Result: runs whose final module passed all evaluations of the time, read from the run summaries (run.toml, summary.toml) of the archived runs in spotforecast2-safe under regen/runs/.

phase	result	why the next phase
1a	29 of 29 evaluations, first test run	blindness rested on the prompt, and it was a single run
1b	3 of 3 runs	the orchestrating session hit the usage limit of the Claude plan
2	stopped: no module written in two attempts	suspected, not tested: the reasoning mode and a harness built for Claude
3	Qwen3.8-27B 3 of 3, Qwen3-Coder 0 of 3	why did Qwen3-Coder fail: no reasoning mode, or only 3B active parameters?
4	Qwen3.8-27B 3 of 3, Devstral 1 of 3 (at T 0.15: 0 of 3)	none, under the decision rule set before the runs the comparison is undecided

3.4.1 Phase 1a: Claude, blind by request

In phase 1a, Claude Opus 5.5 rebuilt the module in a separate worktree, as a subagent of the orchestrating Claude Code session. Its prompt forbade every other source (the old file, its unit tests, the history, the KB, the downstream repositories) and asked it to iterate until the evaluations passed and to report every place where the bundle was ambiguous, silent or contradictory.

The rebuild passed all 29 evaluations of the time on its first test run and failed 3 of the 18 old unit tests. The more important result was the list of eleven such places. Three of them were safety questions (a leak behind an unreadable model, a missing surface, an empty target list) that the old code answered leniently, so that a raw ENTSO-E column could reach a submission with only a warning in the log. The maintainer answered all three with the decision record leakage-guard-fail-closed, which added evaluations for these cases and made the guard fail closed (commit 872ec05, 2026-09-30). The generated module was archived unchanged, as evidence.

3.4.2 Phase 1b: Claude with enforced blindness

Phase 1b addressed the two weaknesses of phase 1a: its blindness rested on the prompt and it was a single run. The script regen_blind.py, the predecessor of spotregen claude, started the Claude Code command line interface (CLI) as claude -p with --restricted and --safe-mode, in a workspace that held only the bundle and

`exceptions.py`. It offered the generator no shell. Before the runs, a canary test (Definition 2.34) showed that the harness refused every attempt to read the original module or to leave the workspace.

Each of the three runs wrote a module that passed all 37 evaluations of the fail-closed specification in its first session, within 2 to 3 minutes. The generators listed 16, 13, and 14 gaps but raised no new safety question. All three stumbled over the same open gaps (the private helper, the type of the forbidden set, the outdated module docstring) and failed closed where the bundle was silent. A gap that recurs across independent sessions points to the specification, not to one session of one model.

Why the generator moved to a local model. The blind runs were cheap (Table 26), but the Claude Code session that orchestrated the study, which researched the sources, wrote the scripts, started the runs, and drafted this report, was not. By the end of phase 1b it had made 196 requests and consumed 43.3 million tokens, 99 % of them cached input, because every request re-sends the growing conversation (on average about 220 thousand tokens). These tokens count against the usage limit of the Claude Max plan. The session hit it. A generator that depends on such a plan cannot be rerun at will for every unit and release, so phases 2 to 4 moved the generator to the maintainer’s laptop. The orchestration stayed in Claude Code.

Table 26: Token consumption of Claude Opus 5.5. Requests: model turns (for the orchestrating session: requests to the hosted model). Cached: input read from the prompt cache. Fresh: input written to the cache or sent uncached. Output includes thinking. Generator rows are read from the archived transcripts of phase 1b. The orchestrating session was measured from its log while it was still running, so its numbers are a lower bound.

	requests	cached input	fresh input	output	thinking	total
canary	7	5.8 K	3.6 K	1.6 K	0.2 K	11.0 K
blind run 1	6	86.3 K	43.9 K	14.8 K	5.6 K	144.9 K
blind run 2	7	83.8 K	41.0 K	12.0 K	3.5 K	136.8 K
blind run 3	6	88.7 K	45.8 K	17.3 K	7.9 K	151.8 K
orchestrating session	196	42.83 M	371.4 K	145.0 K	–	43.35 M

3.4.3 Phase 2: a local model in the Claude Code harness

Phase 2 repeated the protocol of phase 1b with a local model, Qwen3.8-27B, a dense open-weight model of 27 billion parameters, served by LM Studio on the maintainer’s laptop. Only the model changed: `regen_blind.py` pointed Claude Code at the local endpoint (`--base-url`). Bundle, prompt, confinement, and evaluations stayed those of phase 1b.

The phase has two findings. First, the confinement does not depend on the model: the canary test passed unchanged, the harness refused every attempt to leave the workspace, and no request left the machine. Second, the local model wrote no module. In two attempts and 59 minutes it read the bundle and `exceptions.py` and then reasoned in turns of up to 45 thousand tokens, two of which hit the output budget of the harness. Even with the budget raised to 120 thousand tokens no turn ended in a file. The experiment was stopped. In the same harness Claude had needed 2 to 3 minutes and 3.5 to 8 thousand reasoning tokens.

The result does not show that the local model cannot rebuild the guard, only that it cannot do so under the conditions that sufficed for Claude. Three causes are plausible and were not tested separately: the reasoning mode, which spends the output budget on thinking, the harness, whose system prompt and tools were written for Claude models, and the throughput of a 27-billion-parameter model on a laptop. The harness is a realistic cause, since Lewis (2026) raised the number of solved SWE-bench Verified tasks of this very model from 32 to 54 of 169 by changing only the configuration of its harness.

3.4.4 Phase 3: local models in short sessions

Phase 3 applied the architecture of Section 2.9.1 on the same day to the same unit and bundle (the SHA-256 recorded in every `run.toml` is that of the Claude runs) with two local models: Qwen3.8-27B with its reasoning off, and Qwen3-Coder-30B, a mixture-of-experts (MoE) coding model with 3 billion active parameters and no reasoning mode. Each model ran the canary and then three regenerations with seeds 1 to 3 (Table 27).

Both canaries passed on the driver’s logs: no file was created and the only remote address was `127.0.0.1:1234`. Qwen3-Coder nevertheless reported a “Permission denied” error for each requested action, an error that no tool could have produced because none was offered. A model’s account of its own confinement is no evidence.

Table 27: Regenerations of the leakage guard in short local sessions (phase 3), with the three Claude runs of phase 1b for comparison. Evaluations: passed after each attempt. Unit tests: old unit tests passed by the final module, a second opinion (Definition 2.3). Sessions: fresh model requests, including the gap report. Input and output: tokens summed over the sessions, for Claude the cached and fresh input of the whole run. Max.: the largest input of a single session. Time: model time. Gaps: items of the gap report.

run	evaluations	unit tests	sessions	input	output	max.	time	gaps
Qwen3.8-27B 1	31, 32, 37	18/18	4	85.6 K	14.3 K	23.9 K	9.8 min	12
Qwen3.8-27B 2	37	18/18	2	38.1 K	5.7 K	20.7 K	3.7 min	15
Qwen3.8-27B 3	37	18/18	2	37.5 K	4.1 K	20.1 K	2.8 min	8
Qwen3-Coder 1	28, 29, 29	15/18	4	77.9 K	10.8 K	21.3 K	3.3 min	18
Qwen3-Coder 2	32, 32, 32	17/18	4	77.4 K	10.3 K	20.9 K	3.1 min	15
Qwen3-Coder 3	30, 30, 30	17/18	4	78.7 K	9.5 K	21.7 K	3.0 min	20
Claude 1	37	–	1 (6 turns)	130.2 K	14.8 K	–	2.1 min	–
Claude 2	37	–	1 (7 turns)	124.8 K	12.0 K	–	1.7 min	–
Claude 3	37	–	1 (6 turns)	134.5 K	17.3 K	–	2.5 min	–

With its reasoning switched off, Qwen3.8-27B, which had written nothing in an hour in phase 2, passed all 37 evaluations in all three runs, twice on the first attempt and once after two repairs, whose failing evaluations were cases of the fail-closed rule. All three final modules also passed the 18 old unit tests. No session reached its output budget or read more than 24 thousand input tokens. A run took 3 to 10 minutes of model time, against 2 to 3 minutes for Claude. Because the harness and the reasoning mode changed together, the phase does not tell which of the two made the difference.

Qwen3-Coder produced two to three times as many tokens per minute, but no run passed. Its three runs ended at 29, 32 and 30 of 37 evaluations. The failures were of one kind. When a surface could not be read, the module returned silently or named the wrong surface, where the intent items require a `RuntimeError` that names it. This is the fail-open behaviour that `leakage-guard-fail-closed` forbids. The evaluations caught it every time. Repairs rarely helped. Of the six repair sessions, one improved a module (from 28 to 29 passing evaluations) and three returned the previous module byte for byte: with a fixed seed and a prompt that changes only in its failure output, a repair session can reproduce its input.

A local run consumed 42 to 100 thousand tokens, against 137 to 152 thousand for a Claude run, whose single session re-sends its growing history with every turn. No request left the machine. The saving is modest, because a blind run was already short. More telling is the comparison of the two local models: in these runs the faster model without reasoning failed. Only the reasoning model, run with its reasoning off, rebuilt the safety-relevant behaviour from the specification.

3.4.5 Phase 4: Devstral against Qwen

Phase 4 asked why Qwen3-Coder had failed: for lack of a reasoning mode, or because only 3 billion of its parameters are active? It compared Devstral Small 2, a dense model of 24 billion parameters without a reasoning mode (Mistral AI 2025), with Qwen3.8-27B, three runs each under the driver of phase 3. The driver gained two options, a prompt template for Mistral’s chat format and a repair seed that grows with the attempt. A further series ran Devstral at the temperature of 0.15 that its model card uses (Mistral AI 2025). The design and the rule for reading its outcome were fixed before the runs.

Definition 3.9 (Decision rule). A decision rule states, before any run, which outcome of a comparison counts as a difference between two configurations. Setting it in advance prevents the threshold from being chosen after the outcome is known. Phase 4 used the 3:0 rule: with three runs per model, a difference counts only if one model passes all three runs and the other passes none. Every other split, 3:1 included, counts as undecided. The 3:0 split is the most extreme outcome three runs can give. Even it is weak evidence: if both models were equally good, the three passes would fall on one model by chance with probability $1/\binom{6}{3} = 0.05$.

Both canaries passed. Qwen3.8-27B again passed in all three runs, Devstral in one of three, and at temperature 0.15 in none. By the 3:0 rule (Definition 3.9) the comparison is undecided, though the outcomes point weakly in one direction: Devstral failed on the same fail-closed cases as Qwen3-Coder, only less often. The number of active parameters is therefore not the whole explanation. Three runs per model cannot say whether training for reasoning or another difference between the model families accounts for the rest. The growing repair seed did not stop repeats either: in two of Devstral’s runs the second repair returned its input byte for byte.

3.4.6 Redoing the phases

The generation itself cannot be repeated bit for bit, but everything that judges it can. Every run directory keeps the modules, the bundle or its hash, the prompt, the gap report, and the transcript or session log. Its run summary holds the SHA-256 of each module. The hashes can be recomputed, the bundle can be rebuilt at the run’s base commit (Section 3.3), and any archived module can be scored against the evaluations of any commit with `spotregen claude --score`, as this report does while it renders (Table 28).

A new generation run is a new sample, not a replay. A hosted model gives no control over sampling and may change behind its identifier. Claude Code updated itself from 2.1.285 to 2.1.286 between the canary and the attempts of phase 2. For the local phases the weights are pinned by their Hugging Face revision and the sampling parameters and seeds are recorded, but whether LM Studio returns the same tokens for the same seed was not tested. Some runs used script options that were not yet committed (the base field of each record says which), all of which `spotregen claude` and `spotregen local` now offer.

Running the example. In the clone of Section 3, the commands below check a module against its recorded hash, score it, and show the generator command of phase 1b. The last two start a new generation, which needs a logged-in Claude Code or LM Studio with the pinned model loaded:

```
R=regen/runs/2026-09-30-multitask.guards-local-sessions/qwen3.8-27b/run-3
shasum -a 256 $R/guards-1.py # 1bc47b5519641947...: sha256 in $R/run.toml
uv run spotregen claude multitask.guards --score $R/guards-1.py --out score
# evaluations today: 40 passed, 4 failed
uv run spotregen claude multitask.guards --dry-run \
  --allow src/spotforecast2_safe/exceptions.py --model claude-opus-5-5
uv run spotregen claude multitask.guards --attempts 3 --out run-1b \
  --allow src/spotforecast2_safe/exceptions.py --model claude-opus-5-5
uv run spotregen local multitask.guards --model qwen3.8-27b --seed 1 \
  --allow src/spotforecast2_safe/exceptions.py --out run-3
```

3.4.7 Dependents and downstream consumers

The rebuilds above were judged only inside the module, so the blast radius of the deletion test (Definition 2.22) had been checked only statically. Fowler (2027, ch. 2) asks of every component “what breaks if this disappears, and how would we know?”. In chapter 5 he proposes consumer-driven contract testing to record what a consumer needs. On 2026-10-02 the dependents of the guard were therefore checked in three ways. The declared consumers were read for what they rely on. Two needs without an evaluation, the frame of the leak message and the cause in the message for an unreadable estimator, became intent through the decision record `leakage-guard-message-contract` (LG-2, LG-4). `spotregen` ran the dependents inside the repository, the re-export in `multitask/__init__.py` and the docstring examples. A staging run executed the consumers in other repositories, each with its own interpreter, against every candidate module on stub pipelines that are clean, leak on one surface or cannot be read. Table 28 scores the original and the archived rebuilds again.

Table 28: The original guard and the seven archived rebuilds that passed every evaluation of their day, scored again while this document renders with `spotregen claude multitask.guards --score`. Evaluations: today’s durable evaluations. Failing: the evaluations a module fails today. Unit tests: the old unit tests, a second opinion (Definition 2.3). Dependents: the re-export in `multitask/__init__.py` and the docstring example of `assert_no_leakage`. LS: short local sessions (Section 3.4.4), DV: Devstral against Qwen (Section 3.4.5).

module	evaluations	failing	unit tests	dependents
original	44/44	–	18/18	2/2
LS qwen3.8-27b run-1	40/44	<code>test_unreadable_message_states_</code> <code>the_cause</code>	18/18	2/2
LS qwen3.8-27b run-2	40/44	<code>test_unreadable_message_states_</code> <code>the_cause</code>	18/18	2/2
LS qwen3.8-27b run-3	40/44	<code>test_unreadable_message_states_</code> <code>the_cause</code>	18/18	2/2
DV qwen3.8-27b run-1	40/44	<code>test_unreadable_message_states_</code> <code>the_cause</code>	18/18	2/2
DV qwen3.8-27b run-2	40/44	<code>test_unreadable_message_states_</code> <code>the_cause</code>	18/18	2/2

module	evaluations	failing	unit tests	dependents
DV qwen3.8-27b run-3	40/44	test_unreadable_message_states_ the_cause	17/18	2/2
DV devstral-small-2 run-2	40/44	test_unreadable_message_states_ the_cause	18/18	2/2

Inside the repository, every module passes its dependents. Only the new evaluation `test_unreadable_message_states_the_cause` separates them: all seven rebuilds fail it. A promise that was missing from the specification is now a contract that every future rebuild is judged by. Outside the repository, the staging run (432 calls, before and after the fix) found what no check inside could see. Every candidate aborted every downstream run on a leak, but every consumer submitted anyway when the guard could not read a surface: each wrapper caught the `RuntimeError`, logged “leakage guard skipped” and went on. The fail-closed decision ended at the boundary of the package. After the wrappers in four repositories were fixed, every unreadable case aborted. The clean pipelines of all four backends still passed.

4 Discussion

4.1 What the application showed

The preconditions of the deletion test became a question the repository can answer mechanically. For the four pilot units the answer is yes, in the precise sense of *spotregen*: every promise has an evaluation, every dependent inside the repository is declared, and every rationale has resolvable evidence and a review date. The test itself was run for one unit. A generator that had never seen the code rebuilt the leakage guard so that it passed every evaluation of its day, yet all archived rebuilds fail an evaluation added afterwards (Table 28). Evidence has a shelf life: “A module is not correct because it passed the suite in March” (Fowler 2027, ch. 5).

The more interesting result is what the method found. Writing the pilot specifications surfaced latent issues that the code had carried silently. The scoring functions skip a NaN forecast in some metrics while counting it in others, the scaled error lags by rows while the seasonal-naive benchmark lags by calendar offset, and the heal policy of the corruption detector has an anchor zone without an upper bound. These are recorded as known gaps in the invariants of the intent items, so a regenerated module may not silently change them. They are candidates for `fix` commits whose intent items are tightened first. Other findings, such as the two different default contamination values of the outlier helpers, are pinned as current behaviour and justified by a decision record. The first regeneration then turned an implicit behaviour into three explicit safety questions. Answering them changed the leakage guard from lenient to fail-closed. The value of regeneration is therefore not the regenerated file, which is archived only as evidence, but the gaps it exposes in the specification.

The primitives also changed how a behaviour change is made. Because the static check names every promise without an evaluation, the specification-first order is enforced rather than recommended: an intent item without evaluations fails the static check before every release. The requirement that new evaluations fail on the old code gives a simple test of whether a change was really pinned.

4.2 What the measures say

The three measures answer different questions. None of them answers the question of the deletion test alone. Intent coverage (Definition 2.31) is complete for all four units: every written promise has a judge. The mutation score (Definition 2.32) depends on who plants the mutants. On the 42 mutants planted by hand it is 100 %. On the 345 mutants generated by *spotregen* `mutate` it is 82 %. Only the leakage guard keeps 100 % (Table 24). The hand-planted sample measured whether the evaluations catch the bugs their authors expected. The generated sample found what they did not expect: 28 gaps, among them input checks that the fail-safe convention requires but no evaluation could tell from a library error raised later, and thresholds never evaluated at the boundary. A mutation score measured on mutants planted by the authors of the evaluations therefore likely overstates their strength. Complete coverage and a high mutation score are necessary for a positive verdict, but they measure the specification against itself. Neither can see a promise that was never written down. The study found such promises twice: the three lenient gaps of the first regeneration (Section 3.4.1) and the message contract that the staging of the dependents exposed (Section 3.4.7). Each time the coverage stayed complete while the specification grew. Complete coverage and a high mutation score are therefore an entry condition for regeneration, not evidence that the specification is complete. That evidence comes only from attempts: memoryless rebuilds and runs against the dependents.

Specification size (Definition 2.33) measures the price of replaceability, which Fowler (2027, ch. 2) says is bought “boundary by boundary, and the bill comes due before the benefit does”. The knowledge outside the implementation

is about 1.9 times as large as the implementation (Table 23). Most of it is evaluation code, in line with the observation of Fowler (2027, ch. 5) behind Definition 2.33. Each gap that an attempt found added intent items, decision records and evaluations, so the size also records what the repository has learned about its units.

4.3 Choosing a local generator

The two local models of Section 3.4.4 were chosen because they were installed, not because they were the best candidates. The attempts of Section 3.4.3 suggest four criteria for a better choice: reasoning that can be switched off or bounded, training for agentic coding with tools, throughput on the maintainer’s laptop (an M5 Max with 128 GB of unified memory), and weights that fit into that memory with room left for the context. A survey of open-weight models on 2026-09-30 found five that meet most of them (Table 29). The scores are those reported by the vendors and were not checked independently.

Table 29: Candidate local generators for the M5 Max with 128 GB, as of 2026-09-30. Vendor scores: from the model cards.

model	architecture	reasoning	vendor scores	fit and speed
Qwen3-Coder-Next (Qwen Team 2026c)	MoE, 80 B total, 3 B active	none	SWE-bench Verified 70.6, Terminal-Bench 2.0 36.2	8-bit fits; 77 to 79 tokens/s at short context on an M5 Max, 48 at 66 K (oMLX 2026; Silicon Score 2026)
gpt-oss-120b (OpenAI 2025)	MoE, 117 B total, 5.1 B active	low, medium or high, set in the system prompt	SWE-bench Verified 62.4 at high reasoning	about 65 GB in MXFP4, a 4-bit microscaling floating-point format
Qwen3.6-35B-A3B (Qwen Team 2026a)	MoE, 35 B total, 3 B active	on by default, switchable (enable_thinking)	SWE-bench Verified 73.4, Terminal-Bench 2.0 51.5	fits easily
Devstral Small 2 (Mistral AI 2025)	dense, 24 B	none	SWE-bench Verified 68.0, Terminal-Bench 2.0 22.5	fits easily
Qwen3.8-Flash-Next (Qwen Team 2026b)	MoE, 125 B plus a 51 B n-gram embedding, 6 B active	on by default, switchable	SWE-bench Pro 62.5	112 GB at 4 bits (LLMCheck 2026), which leaves about 16 GB for the context

Larger open-weight models do not fit. The 4-bit files of MiniMax-M3 (428 B total, 23 B active) need 208 to 265 GB (Unsloth AI 2026), those of Kimi-K2.6 312 GB (Meguedmi 2026), and GLM-5.3 needs 239 GB even at 2 bits (LLMCheck 2026). Its smaller sibling GLM-5.3-Flash (320 B total, 18 B active) needs 120 GB at 3 bits (LLMCheck 2026), which leaves almost no room for the context. DeepSeek-V4-Flash (284 B total, 13 B active) fits only at 2 bits (LLMCheck 2026) and formats its prompts with scripts of its own instead of a chat template (DeepSeek-AI 2026). Mistral Medium 3.5, which replaced Devstral 2 in Mistral’s agent, is a dense model of 128 B parameters (Mistral AI 2026) and would be slower than the 27 B model that was already too slow in Section 3.4.3. The harness of Section 3.4.3 needs no change for any candidate, since LM Studio has served an Anthropic-compatible endpoint with tool use since version 0.4.1 (LM Studio Team 2026).

4.3.1 Lessons learned

The phases of the regeneration (Table 19) changed the generator and its harness, while the bundle, the evaluations and the deletion of the module stayed the same. Their results, reported in Section 3.4, lead to four lessons.

First, a memoryless generator has to be enforced and recorded, not requested. In phase 1a Claude Opus 5.5 rebuilt the leakage guard in a single run, but only its prompt kept it from the old module. In phase 1b the harness refused every access outside the workspace, a canary test showed the refusal, and all three runs passed the 37 evaluations in their first session within two to three minutes. Phases 3 and 4 add two pieces of evidence that do not depend on the model, the canary and the remote addresses of the driver’s sockets. The models’ own accounts of their confinement were partly invented and are therefore no evidence.

Second, for a local model the harness matters as much as the model. In phase 2 Qwen3.8-27B ran in the harness of Claude Code with its reasoning on and wrote no module in 59 minutes. In phase 3 the same weights, driven by the

short sessions of Section 2.9.1 with the reasoning switched off, passed all evaluations in three of three runs. Phase 4 repeated this result with a changed driver.

Third, the choice of model decides the safety-relevant cases. Qwen3-Coder-30B passed none of three runs and Devstral Small 2 one of three. Almost all of their failures were cases of the fail-closed rule, in which a module returned silently where the intent items require a `RuntimeError`. The evaluations caught every one of them. Only Qwen3.8-27B, a model trained for reasoning but run with its reasoning off, passed every local run.

Fourth, repairs help less than independent runs. Qwen3.8-27B passed on the first attempt in three of its six runs and after one or two repairs in the other three. For Qwen3-Coder and Devstral only one of ten repairs improved a module. Five returned their input byte for byte, with a fixed seed and with a seed that grows with the attempt alike. Three runs with different seeds and a bounded number of repairs therefore give more information than a long repair loop.

4.3.2 Design recommendation

If the study were done again from scratch, we would start directly with the architecture of Section 2.9.1 and skip phases 1a and 2. The generator would be Qwen3.8-27B with its reasoning off, driven by `spotregen local`. If affordable, three runs of Claude Opus 5.5 under `spotregen claude` would serve as a hosted reference with the settings of phase 1b. Every comparison between models would follow a decision rule set before the runs (Definition 3.9). The runs would use a released version of `spotregen`, so that the base field of every record names a clean commit rather than options that were not yet committed (Section 3.4.6). Whether LM Studio returns the same tokens for the same seed was not tested in this study and should be tested before the first run. Finally, the session that orchestrates the study should itself be split into short sessions. Table 30 lists the recommended tools and settings, which are those of the Qwen3.8-27B runs of phase 4.

Table 30: Recommended setup for a regeneration, derived from the five phases of Table 19. Evidence: the phases whose results support the setting.

component	recommended tool or setting	evidence
hardware	Apple M5 Max, 128 GB unified memory, macOS 27.0	phases 2 to 4
model server	LM Studio headless (llmster), MLX engine 1.11.0, OpenAI-compatible completion endpoint on 127.0.0.1:1234, context length 32,768 tokens	phases 3 and 4
generator	Qwen3.8-27B, 4-bit MLX (mlx-community/Qwen3.8-27B-4bit, revision 10c35ca), reasoning off by an empty reasoning block written into the prompt	6 of 6 runs passed
driver	<code>spotregen local</code> of <code>spotregen 0.10.1</code> , standard library only, no tools, short sessions (Figure 10)	phases 3 and 4
inputs	the bundle of <code>spotregen context</code> and <code>exceptions.py</code> as the only allowed file	all phases
sampling	temperature 0.7, top-p 0.8, top-k 20, seed equal to the run number, repair seed growing with the attempt	phase 4
budgets	3 attempts, 8,192 output tokens per code session, 4,096 for the gap report, 15 minutes per session	phases 3 and 4
confinement	the canary (<code>--canary</code>) before the runs, endpoints only on localhost, socket log by <code>lsof</code>	phases 1b to 4
judges	the durable evaluations; the old unit tests and the dependents as a second opinion; run by <code>spotregen local</code> itself (by <code>spotregen claude --score</code> for archived modules), and the maintainer for safety decisions	all phases
hosted reference	Claude Opus 5.5 through <code>spotregen claude</code> , i.e. <code>claude -p</code> with <code>--restricted</code> and <code>--safe-mode</code>	phase 1b
repetitions	three runs per model, with the design and a decision rule (Definition 3.9) set before the runs	phase 4

In a clone of `spotforecast2-safe` with LM Studio installed, the recommended setup runs as follows (the last command runs the hosted reference):

```
lms load qwen3.8-27b --context-length 32768 --gpu max
uv run spotregen context multitask.guards -o bundle.md
```

```

uv run spotregen local multitask.guards --model qwen3.8-27b --canary \
    --allow src/spotforecast2_safe/exceptions.py --out canary
for s in 1 2 3; do
    uv run spotregen local multitask.guards --model qwen3.8-27b --seed $s \
        --repair-seed step --allow src/spotforecast2_safe/exceptions.py --out run-$s
done
uv run spotregen claude multitask.guards --attempts 3 --out claude \
    --allow src/spotforecast2_safe/exceptions.py --model claude-opus-5-5

```

4.4 Limitations and threats to validity

Several limitations qualify the results. First, the evaluation of regeneration rests on a single module. It was rebuilt four times by Claude Opus 5.5 and fifteen times by three local models (Qwen3.8-27B six times, Qwen3-Coder-30B three times, and Devstral Small 2 six times at two temperatures), with one seed per run. These runs show that a rebuild from the specification is feasible and repeatable, not how often it succeeds across units or models. A systematic study would regenerate every unit several times with several models and report the distribution of outcomes and gaps. The gap reports of the local runs were not classified by topic. The blindness of phase 1a rested on the prompt and was verified only afterwards, from the tool-call log. The canary of phase 1b shows the confinement of Claude Code 2.1.285 only. The attempts of phase 2 ran on 2.1.286, to which the CLI had updated itself in the meantime. Phase 2 is a single configuration of a single model and was stopped early, a negative observation rather than a measurement. Phase 3 changed the harness and the reasoning mode at once and so does not separate the two causes. The comparison of phase 4 remained undecided with three runs per model. The driver was corrected twice during the study. The two affected runs were discarded and repeated, as recorded in the archive.

Second, the mutation score (Definition 2.32) shows that the evaluations would notice a change, not that they test the right thing (Fowler 2027, ch. 5). A promise that nobody wrote down is not protected by any number of killed mutants. Regeneration partly addresses this, because the old unit tests act as an independent second opinion, but it cannot find a promise that neither the specification nor the old tests recorded. The mutants of the pilot were also few and were planted by the same LLM agents that wrote the evaluations, so the two are not independent. The generated sample of `spotregen mutate` (Table 24) does not depend on that choice, but it has a bias of its own: each operator changes one node, so a bug that spans several statements is not in the sample. An equivalent mutant counts as a survivor until a person has looked at it.

Third, external consumers are declared, not checked by the static check. The check cannot see a downstream script that starts to depend on a unit, so the declared blast radius is only as current as its maintainer keeps it. Grepping the downstream repositories is a manual step of the workflow. The staging run of Section 3.4.7 executed the downstream consumers of one unit once, with stub pipelines instead of trained models. It is an archived experiment, not a recurring check. Consumer-driven contract testing would make it one (Fowler 2027, ch. 5): LG-2 and LG-4 record two consumer needs on the provider side, but the downstream repositories do not yet record them themselves. Operational and human consumers cannot be run at all; only the evaluations of the needs they rely on protect them.

Fourth, much of the specification was reconstructed from code, tests, and KB entries by an LLM agent and then reviewed. Such records are marked with `origin = "machine"`, but a reconstructed rationale can be plausible and wrong. The review dates limit how long such an error can persist without being questioned. They do not prevent it.

Fifth, pace was recorded but not tested. `spotregen` copies the verification a pace class requires into the bundle but does not enforce it. All four pilot units are high-risk, so the pilot cannot show whether the classes earn their cost. “Pace without evidence becomes bureaucracy” (Fowler 2027, ch. 3).

Finally, the study has one author, who is also the maintainer of the library and the person who approved the decisions. The measurements are reproducible from the repository, but the judgements (which units are high-risk, which behaviours are promises) are not independent.

4.5 Transfer

Nothing in `spotregen` is specific to forecasting. The schema and the checks were first built for `macl2l` and generalised through `architecture.toml` (scan directories, documentation globs, KB location, known cycles). The natural next candidates in `spotforecast2-safe` are the resilient downloader, the imputation helpers, and the audit-log manager, whose log schema is a slow contract by the project’s release rules. The sibling library `spotforecast2` consumes three of the four pilot units and would profit from declaring its own consumers of them.

5 Conclusion

The introduction promised to describe how the seven primitives of Fowler (2027) were applied to `spotforecast2-safe` and what the application produced. Each primitive became a record or a check in the repository. The attempt to delete a module decided whether the records were enough. Intent lives in the unit records as items with a scope, an invariant, and a reason. The negative constraints among them record what a competent stranger would not think of (Section 2.3.1). Compilation is the import policy of `architecture.toml`, recorded as a baseline of the tree rather than as a design target, with the existing cycles as debt that can only shrink (Section 2.3.2). Evaluation is one `pytest` file per unit whose tests carry the marker `regen` and judge any implementation through the public boundary (Section 2.3.3). Provenance is one decision record per rationale, with resolvable evidence, assumptions, a review date, and an origin that separates a person’s decision from a reconstructed one (Section 2.3.4). Pace is a class per unit and a verification policy per class (Section 2.3.5). Deletion is the list of consumers per unit together with the subcommand that checks the preconditions of the deletion test without deleting anything (Section 2.3.6). Compaction is the subcommand that reports the records the code has outgrown (Section 2.3.7). The static check ties the seven together: a promise without an evaluation, an import across a boundary, evidence that does not resolve or an undeclared dependent stops a release (Section 2.4). The regeneration bundle is the product of these records. `spotregen context` builds it from them without a language model.

Does the approach work? Applied to a safety-critical library, the seven primitives of Fowler (2027) turn the preconditions of the deletion test into a static check. Regeneration turns the test itself into a repeatable experiment. Intent, provenance, and consumers live in small reviewed records, evaluations judge any implementation through the public boundary, and the standard-library package `spotregen` checks the whole arrangement before every release. The pilot shows that four high-risk modules of `spotforecast2-safe` can be specified this way, that every written promise has an evaluation (complete intent coverage), that the evaluations kill every mutant planted by hand and 82 % of the mutants generated by `spotregen mutate`, that they accepted the memoryless rebuilds of the leakage guard on the day of each run, and that the process finds real gaps: a missing message contract and three safety questions that the code had answered leniently. The specification also carries a rebuild without any hosted model: in short, tool-less sessions on a laptop, a local model with its reasoning switched off passed every evaluation in three of three runs. For the leakage guard, the knowledge that used to live only in the code, and in the memory of its maintainer, has survived the deletion of the code. For the other three units, only an attempt can show it.

The answer is positive where a primitive has both a record and a check, and open where it has only a record. Pace was recorded but never exercised, because all four pilot units are high-risk and `spotregen` does not enforce the verification a class requires. Compaction sees the records in `regen/` and not the library, so it finds stale knowledge but no obsolete code. External, operational, and human consumers are declared but cannot be checked from inside the repository. The staging run that executed the downstream consumers was done once, not as a recurring check. The deletion test proper was run for one unit. Every archived rebuild that passed on its day fails an evaluation added afterwards, so a verdict holds for the specification of its day only. The price is a specification about 1.9 times the size of the implementation, most of it evaluation code. Within these limits the primitives did what Fowler (2027) asks of them: the knowledge of the leakage guard outlived its code, the attempt found gaps that no measure had shown, and the static check keeps the records in step with the code before every release.

Data and code availability

All artefacts are part of the `spotforecast2-safe` repository (Bartz-Beielstein 2026b), version 29.2.1, commit 41c0e38: the knowledge in `regen/`, the evaluations `tests/test_regen_eval_*.py`, the archived regeneration runs in `regen/runs/` and the guide `HOWTO-REGEN.md`. The checks are run by the package `spotregen` (Bartz-Beielstein 2026d), version 0.10.1, on PyPI. The regeneration runs in `regen/runs/` were made with its predecessors, the scripts `scripts/regen.py`, `scripts/regen_blind.py` and `scripts/regen_local.py` of `spotforecast2-safe`, the staging run with its own standard-library harness, and the mutation runs with `spotregen 0.4.0` (`regen/runs/2026-10-02-mutation`, with `review.toml`, and `regen/runs/2026-10-02-mutation-pinned`). This report and its German version are kept in the repository `bart26x-regen`. Running `make render` there re-executes every example against a checkout of `spotforecast2-safe`.

Use of AI tools

Transparency notice: parts of this report were researched, drafted, and translated with the support of artificial intelligence. This information is provided voluntarily for complete transparency, as no legal requirement mandates its disclosure.

References

- Amat Rodrigo, Joaquin, and Javier Escobar Ortiz. 2026. *Skforecast*. V. 0.23.0. Released. <https://doi.org/10.5281/zenodo.8382787>.
- Barr, Earl T., Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. “The Oracle Problem in Software Testing: A Survey.” *IEEE Transactions on Software Engineering* 41 (5): 507–25. <https://doi.org/10.1109/TSE.2014.2372785>.
- Bartz-Beielstein, Thomas. 2026a. *spotforecast2: Time-Series Forecasting with Sequential Parameter Optimization*. <https://gitlab.git.nrw/thk-f10/spotsevenlab/spotforecast2>.
- Bartz-Beielstein, Thomas. 2026b. *spotforecast2-safe: Safety-Critical Subset of spotforecast2*. <https://git-ce.rwth-aachen.de/thomas.bartz-beielstein/spotforecast2-safe>.
- Bartz-Beielstein, Thomas. 2026c. *spotregen: Regenerative Software Tooling*. Python Package Index (PyPI). <https://pypi.org/project/spotregen/>.
- Bartz-Beielstein, Thomas. 2026d. *spotregen: Tooling for Regenerative Software*. <https://git-ce.rwth-aachen.de/thomas.bartz-beielstein/spotregen>.
- Bartz-Beielstein, Thomas, and Eva Bartz. 2026. *Time-Series Forecasting in Safety-Critical Environments: An EU-AI-Act-Compliant Open-Source Package / Zeitreihenprognose in sicherheitskritischen Umgebungen: Ein KI-VO-konformes Open-Source-Paket*. <https://doi.org/10.48550/arXiv.2604.23859>.
- Brand, Stewart. 1994. *How Buildings Learn: What Happens After They’re Built*. Viking.
- Cannon, Brett, Nathaniel J. Smith, and Donald Stufft. 2016. *PEP 518 – Specifying Minimum Build System Requirements for Python Projects*. Python Enhancement Proposals. <https://peps.python.org/pep-0518/>.
- DeepSeek-AI. 2026. *DeepSeek-V4-Flash*. Model card, Hugging Face. <https://huggingface.co/deepseek-ai/DeepSeek-V4-Flash>.
- DeMillo, Richard A., Richard J. Lipton, and Frederick G. Sayward. 1978. “Hints on Test Data Selection: Help for the Practicing Programmer.” *Computer* 11 (4): 34–41. <https://doi.org/10.1109/C-M.1978.218136>.
- European Parliament and Council. 2024. *Regulation (EU) 2024/1689 of the European Parliament and of the Council of 13 June 2024 Laying down Harmonised Rules on Artificial Intelligence and Amending Regulations (EC) No 300/2008, (EU) No 167/2013, (EU) No 168/2013, (EU) 2018/858, (EU) 2018/1139 and (EU) 2019/2144 and Directives 2014/90/EU, (EU) 2016/797 and (EU) 2020/1828 (Artificial Intelligence Act)*. Official Journal of the European Union, L series. <https://eur-lex.europa.eu/eli/reg/2024/1689/oj>.
- Fowler, Chad. 2027. *Regenerative Software*. O’Reilly.
- Hukkinen, Taneli, and Shantanu Jain. 2022. *PEP 680 – Tomllib: Support for Parsing TOML in the Standard Library*. Python Enhancement Proposals. <https://peps.python.org/pep-0680/>.
- Jia, Yue, and Mark Harman. 2011. “An Analysis and Survey of the Development of Mutation Testing.” *IEEE Transactions on Software Engineering* 37 (5): 649–78. <https://doi.org/10.1109/TSE.2010.62>.
- Karpathy, Andrej. 2025. *Software Is Changing (Again)*. Andrej Karpathy. <https://youtu.be/LCEmiRjPEtQ?si=uytPlpy7M3wTESW1>.
- Lewis, Sydney. 2026. *Same Model, Different Harness: Different Coding-Agent Results*. <https://arxiv.org/abs/2608.26218>.
- LLMCheck. 2026. *State of Open-Source Local LLMs – September 2026: Two Frontier MoEs Reach the Mac Studio*. <https://llmcheck.net/blog/state-of-open-source-local-llms-september-2026/>.

- LM Studio Team. 2026. *Use Your LM Studio Models in Claude Code*. LM Studio blog. <https://lmstudio.ai/blog/claude-code>.
- Meguedmi, Mohamed. 2026. *Best MLX Local LLMs on Apple Silicon in 2026*. Bestllmfor.com. <https://bestllmfor.com/guides/mlx-local-llm-apple-silicon/>.
- Mistral AI. 2025. *Devstral-Small-2-24B-Instruct-2512*. Model card, Hugging Face. <https://huggingface.co/mistralai/Devstral-Small-2-24B-Instruct-2512>.
- Mistral AI. 2026. *Remote Agents in Vibe. Powered by Mistral Medium 3.5*. Mistral.ai news. <https://mistral.ai/news/vibe-remote-agents-mistral-medium-3-5/>.
- Nygard, Michael. 2011. *Documenting Architecture Decisions*. Cognitect blog. <https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions>.
- oMLX. 2026. *Qwen3-Coder-Next on M5 Max (40c) – oMLX Benchmark*. <https://omlx.ai/benchmarks/performance/k7wfh7wb>.
- OpenAI. 2025. *gpt-oss-120b*. Model card, Hugging Face. <https://huggingface.co/openai/gpt-oss-120b>.
- Parnas, David L. 1972. “On the Criteria to Be Used in Decomposing Systems into Modules.” *Communications of the ACM* 15 (12): 1053–58. <https://doi.org/10.1145/361598.361623>.
- Preston-Werner, Tom, Pradyun Gedam, et al. 2021. *TOML: Tom’s Obvious, Minimal Language, Version 1.0.0*. <https://toml.io/en/v1.0.0>.
- Qwen Team. 2026a. *Qwen3.6-35B-A3B*. Model card, Hugging Face. <https://huggingface.co/Qwen/Qwen3.6-35B-A3B>.
- Qwen Team. 2026b. *Qwen3.8-Flash-Next*. Model card, Hugging Face. <https://huggingface.co/Qwen/Qwen3.8-Flash-Next>.
- Qwen Team. 2026c. *Qwen3-Coder-Next*. Model card, Hugging Face. <https://huggingface.co/Qwen/Qwen3-Coder-Next>.
- Saltzer, Jerome H., and Michael D. Schroeder. 1975. “The Protection of Information in Computer Systems.” *Proceedings of the IEEE* 63 (9): 1278–308. <https://doi.org/10.1109/PROC.1975.9939>.
- Segura, Sergio, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. “A Survey on Metamorphic Testing.” *IEEE Transactions on Software Engineering* 42 (9): 805–24. <https://doi.org/10.1109/TSE.2016.2532875>.
- Silicon Score. 2026. *Qwen3-Coder-Next Mac Benchmark: M5 Max (128 GB), RAM Fit*. <https://siliconscore.com/models/qwen3-coder-next/>.
- Unsloth AI. 2026. *MiniMax-M3-GGUF*. Model repository, Hugging Face. <https://huggingface.co/unsloth/MiniMax-M3-GGUF>.