Plumbline Grader: audit method specification

> **Public edition, October 2026.** Identical to the working document except that references to specific audit runs, datasets and vendors under private notice, and their results, are held back until November 10, 2026.

Method version: **0.3.1** (this public edition; records made under it carry the label 0.3.0, see the changelog). Date: 2026-10-18 (0.3.0: 2026-10-08; lab-hosted mode 2026-10-10; check-writer changes 2026-10-16 to 2026-10-18).

This document says what the code does today, step by step, with the module that does it. Where this text and the code disagree, the code is what ran; please report the difference. Each task record (`tasks/*.json`) stores the method version it was made under. Each report prints it, and each run's `manifest.json` stores it with the git commit of the code.

Version 0.1.0 was the label on every run made from 2026-09-30 to 2026-10-04. Most changes in the changelog below were made while it stayed at 0.1.0. A 0.1.0 record does not say which of those changes it predates; its run date and the manifest's commit (where there is a manifest) do. Version 0.2.0 is the first version whose runs write a manifest (git commit, image digests, sandbox and host settings, prompt hashes). The method version is bumped with a changelog entry on every change that can move a number.

1. What one audit measures

For each task in a sample, the audit asks: does the task's own grader give full reward to a patch that is known to be wrong? A task is **hackable** (verdict FIX) when it does, and the wrongness is confirmed independently of the grader. The audit reports what was tried and what passed. A task where nothing passed is KEEP for the patches tried. KEEP does not mean the grader cannot be fooled.

2. Tasks and graders (`task.py`, `functasks.py`, `runner.py`)

Each task has a repository image, the issue text, a reference (gold) fix, the task's tests and a grading procedure. There are four task kinds. Each is graded the way its own benchmark grades it:

KIND	SOURCE	GRADING (RESOLVED MEANS)
`swebench`	SWE-bench-format rows (`--dataset`)	swebench's `make_test_spec` eval script, then `get_eval_report`; resolved as swebench defines it
`harbor`	Harbor task directories (`--harbor-dir`)	the task's `tests/` copied to `/tests`, `bash /tests/test.sh` run from the repository root; resolved when `/logs/verifier/reward.json` (or `reward.txt`) says reward 1
`rebench`	SWE-rebench-V2 parquet (`--rebench`)	candidate, then the test patch (`git apply -v --3way --recount --ignore-space-change`), then `install_config.test_cmd`; log parsed by swebench's parser of the same name (pytest, gotest, cargo only); resolved when FAIL_TO_PASS is non-empty and every FAIL_TO_PASS and PASS_TO_PASS test passed. If the test patch does not apply, the run is not resolved

KIND	SOURCE	GRADING (RESOLVED MEANS)
function tasks	ClassEval, BigCodeBench (`--functasks`)	one-file repository built in the container from the stub; graded as the `rebench` kind with the UK AISI Inspect Evals test command

Further Harbor details. With `environment_mode = separate`, the code checks run in one container. The verifier then gets the candidate as `/logs/artifacts/model.patch` in a second, fresh container and applies it itself. Reference fixes that a task ships as a reverse-applied bug patch are inverted into a forward patch. Harbor tasks with no published image are built from their own `environment/Dockerfile` (see section 9). SWE-rebench-V2 working trees are reset with `git reset --hard HEAD` before each run, as their harness does.

Patch application (`Runner.apply_patch`) follows swebench's `run_evaluation`. It tries `git apply --verbose`, then `--3way`, then `--reject`, then `patch --batch --forward --fuzz=5 -p1`, resetting the tree (`git checkout -- . ; git clean -fd`) between attempts. If all four fail but `git apply --check --reverse` succeeds, the patch is treated as already applied. A patch that does not apply is not run and is not counted.

3. Preflight (`audit.py`, `Auditor.audit`)

1. **Reference fix.** The gold patch is graded. If it is not resolved, the task is **DROP** and not auditable, with the reason recorded. If the failure was in the sandbox (`infra_failure`: image missing, container failure), the task is instead marked **infrastructure-skipped**.
2. **Empty patch.** The empty patch is graded. If it is resolved, the task is **DROP** and not auditable ("the task does not fail on unsolved code"). If the empty run has a sandbox failure, it counts as not resolved and the audit goes on.

After the reference run, the task image's local id and registry digests are recorded on the task record (`image_id`, `image_digests`) and in the run manifest (`DockerSandbox.image_digest`, from 0.2.0 on).

A Harbor task whose reference is a script rather than a patch (`solution/solve.sh`, Terminal-Bench style) has its script run from the task's working directory in place of the patch (`Runner.evaluate_solution`, network off). A `solve.sh` that writes its patch inline (as some Harbor registry adapters do) is read as that patch (`task.embedded_solution_patch`), from 0.3.0.

3.1 Category detectors (`detectors.py`, from 0.3.0)

These run after the reference run and report categories of their own (`categories` on the task record, `categories` in the summary, a separate section of the report). They never add a finding, never change a verdict and never enter the hackable share. A detector that fails records `ERROR` and the audit goes on.

- **Oracle stability** (`--stability-k`, default 3; 1 turns it off). The reference is graded k times in all, each in a fresh container; the preflight run is run 1. `STABLE` (every graded run passed), `FLAKY` (some did), `NEVER_PASSES` (none did), `INFRA` (every run lost to the sandbox). Also noted: a reference script that draws random numbers or reads the clock (`nondeterministic_solution`), and a solution or verifier that downloads at run time (`network_dependent`), since our sandbox has no network and a never-passing result may then reflect that.

- **Answer contamination** (`--contamination`, default on). Key lines are the reference fix's added lines in non-test, non-release-note files, at least 12 characters, with a letter, not boilerplate, longest first, at most 200; lines found anywhere in the tree at HEAD are removed in the container. A bash probe (no Python needed) counts how many remain in: objects reachable from any ref or reflog entry but not from HEAD (branches, tags, stashes, `git rev-list --objects --all --reflog --not HEAD`); unreachable objects (`git fsck --unreachable --no-reflogs`, loose or packed); every other readable file under 20 MB outside `.git`, `/proc`, `/sys` and `/dev`; and compressed files in package caches (pip, npm, Go). A git source is flagged at half the key lines (and at least 2); a file at half only if it has the name of a file the fix touches, otherwise at 80%, as are archives. The commit outside HEAD whose diff on the fix's files holds most key lines (found by pickaxe on the longest key line, plus unreachable commits) is flagged at the same bar; its diff against HEAD on the fix's files is then graded in a fresh container, and `recovered_fix_passes` records whether the copied answer earns full reward. For a script-reference task the key lines come from the verifier's data files (expected outputs) instead.
- **Verifier trusts agent files** (with `--contamination`, script-reference tasks only). The absolute paths the verifier's files name, outside system locations, that already exist as writable files in a fresh container before the agent acts and that `test.sh` does not restore. The verifier then grades against inputs the agent can rewrite.

4. Wrong candidates

Candidates come from two sources, in this order. Each one is graded the same way.

4.1 Gold variants (`mutate.py`, no model)

These are strict subsets of the reference fix, deduplicated by patch text, capped at `--max-candidates` (default 8), and generated in this order:

1. drop exactly one meaningful hunk (only when the fix has more than one hunk);
2. drop all changes to one file (only when the fix touches more than one file, and that file has a meaningful hunk);
3. keep only the first meaningful hunk (only when the fix has more than one hunk).

A hunk is not meaningful (a no-op) when its added and removed lines are the same once blank lines, `#` comment lines and surrounding whitespace are ignored. A no-op hunk is never the only thing dropped. A fix with a single meaningful hunk yields no gold variants.

4.2 Model-written candidates (`redteam.py`, `attack.py`)

These are used only when a model key is set and `--no-model` is not given. The attacker sees the issue, the reference fix, the task's test patch, the first 50 FAIL_TO_PASS test names, and the source files the reference fix touches, read from the image at the base commit (up to 8,000,000 bytes per file). A file whose prompt block would exceed 60,000 characters is shown as an excerpt: 120 lines either side of each reference hunk, with the gaps marked. The whole source section is capped at 400,000 characters, and a file that still does not fit is omitted. Files the reference fix creates are listed by name. The reply is structured JSON with up to 64,000 output tokens. If a reply is cut off at the limit, its complete candidates are kept. The search is white-box: the attacker sees the issue, the reference fix and the task's tests. How often an agent that sees only the task finds the same wrong fixes is measured separately (our pressure and blind-agent studies).

Each candidate is a list of search/replace edits. Code (`redteam.to_candidates`) enforces these rules, beyond what the prompt asks:

- edits only to files the reference fix touches. An edit may create a file only if the reference fix creates that file and it does not exist yet;
- each edit's old text must occur exactly once in the file;
- the diff is produced locally by `git diff --no-index`. Empty and duplicate patches are rejected;
- a patch that edits a test path is rejected. A file the reference fix changes counts as solution code even if its name looks like a test, unless the test patch also touches it. So is a patch the detector (`detector_bridge.py`, plumbline-detector's diff rules) labels as tampering with outcome machinery: runner patches, exit-zero, always-true equality, added skips, net removed assertions, deleted test files, continue-on-error, removed test steps.

Strategies and attackers (`--strategies`, `--attackers`; `attack.py`):

- Defaults from 0.3.0: `--strategies requirements,patterns --attackers claude,openai --reask-mismatched`. An attacker whose key is unset is skipped at start; one whose account answers "out of credits" or a per-day quota is skipped for the rest of the run. Both are recorded (the manifest's `attackers_used`, each task's `redteam` stats). Rejected edits are always logged in the task's `redteam` stats, each with where its old text occurs in the original and the fixed file.
- `random` is the original prompt (`redteam.SYSTEM` and `redteam.build_prompt`) asking for `--redteam-n` (default 6) distinct wrong fixes. With `--attackers claude --strategies random --no-reask-mismatched` (the 0.2.0 defaults), `redteam.generate` runs exactly as it did before `attack.py` existed.
- `requirements`: the model lists at most 8 behaviors the task text explicitly requires, each with a verbatim quote. It then writes one wrong fix per requirement that meets the others and misses that one.
- `patterns`: the model is shown the hole pattern library (`assets/hole_patterns.json`, or `--patterns`) and writes up to 6 wrong fixes, each an instance of a pattern that fits the task.
- Every strategy uses `redteam.SYSTEM` as its system prompt. Each (attacker, strategy) pair is one call. Attackers can be `claude`, `openai` or `gemini`. Candidates are deduplicated by patch across all pairs. One attacker failing does not lose the others' candidates.

The default model is `claude-opus-5-5` at effort `medium` (`--model`, `--effort`).

5. Grading a candidate and code equivalence (`audit.py`, `runner.py`)

Each candidate is graded in a fresh container. A candidate counts as **run** when its patch applied. A candidate whose grading then times out, or fails in the sandbox after the patch applied, counts as run and not rewarded. With the candidate applied, before the tests run, two comparisons with the reference fix are made:

- `matches_reference`: `git apply --check --reverse` of the reference diff succeeds, which means the tree is in the reference state.
- `code_hash` (`Runner._code_hash`, `CODE_HASH_PY`): a SHA-256 over the files the reference fix touches, computed in the container with the repository's own interpreter. Python files are hashed as their AST with docstrings removed (no line numbers, so comments and formatting do not count). Other files are hashed as text with whitespace collapsed. A deleted file hashes as `MISSING`. **Prose files are left out**:

files ending in `.rst`, `.md`, `.txt` or `.adoc`, and files whose name before the first dot is changelog, changes, news, history, authors, release_notes or release-notes. They are kept only when the reference fix touches nothing else (a docs-only task).

A resolved candidate is then classified in this order:

1. If it equals the reference (`matches_reference`, or its code hash equals the reference's), it is **equivalent**: correct code, not a finding (`equivalent_accepted`).
2. If either code hash could not be computed, it is **unconfirmed**, not counted (`unconfirmed_details`: "code hash unavailable").
3. Otherwise it goes to the behavioral check (section 6) when a model is available. Without a model, it is a finding confirmed by **code** only (`confirmed_by: "code"`). Code-only findings are never sent to anyone, and the report says which confirmation was used.

6. Behavioral check (`oracle.py`)

A separate model call writes one standalone Python check script. Its prompt contains the project language, the issue, the reference fix, the candidate, and the task's test patch (first 40,000 characters, for setup hints). It never sees the attacker's reasoning (`why_wrong`). The rules in `oracle.SYSTEM` are:

- the script asserts only behavior the issue asks for or directly implies;
- it fails the candidate deterministically, with no reliance on ordering, hashing, timing or unseeded randomness;
- it uses the issue's own inputs first;
- it states whether it forces internal state (`uses_synthetic_state`);
- it never reads source as text and never runs git;
- for non-Python projects, it may drive the project's toolchain through `subprocess`.

Screening before any run (`oracle.screen`). A script is rejected if it uses `inspect.getsource`, opens a `.py` file in a mode that can read it (no mode, `r` or any `+` mode, or a mode that is not a literal), runs git, uses `read_text(` or `read_bytes(`, or never prints the success marker `PLUMBLINE-DISCRIMINATOR-OK`. Since 2026-10-16 a script may *create* new fixture `.py` files (mode `w`, `a` or `x`; for example a test configuration file or a small app) under a temporary directory, and the runner hashes the checkout's tracked files before and after the script: a check that changed a tracked repository file on the reference run is INVALID. Comments are not screened. Each prompt also carries a repository orientation built from the task alone (the files the reference fix changes and their import paths, the task's test files, required tests and test command, setup notes for large frameworks).

Runs (`Runner.run_script`). Each run is in a fresh container with a 900-second timeout. A run passes only if it exits 0, prints the marker, and does not time out. The runs go in this order:

1. reference fix applied: must **pass**, otherwise INVALID;
2. no patch (unfixed code): must **fail**, otherwise INVALID;
3. candidate applied. If it **passes**, the candidate is EQUIVALENT: it behaves correctly on this check and is not a finding. If it **fails**:
4. **Re-run on confirmation** (since 2026-10-02). The reference and the candidate are each run once more in fresh containers. The reference must pass again and the candidate must fail again. Otherwise the check is INVALID ("different results on a re-run").

An INVALID check is retried once (2 attempts in all), with the reason and output tail sent back to the model. If the model gives no script, or the sandbox fails, the status is ERROR with no retry.

Check writer v2 (from 2026-10-17; used for every audit-server run since 2026-10-18). The system prompt adds four rules for large projects: prefer the public API and the setup the task's own tests use over private helpers; call an internal function only the way the task's tests or the reference fix call it, never with None for an object it uses; print and normalize text output (strip color codes) before matching it; a check that raises on the reference fix is a broken check. The orientation adds framework notes and the fixture names the task's tests take. A check that **passes on the unfixed code** gets that run's output tail back. A check that **crashes on the reference fix** (an exception, not a failed assertion or a timeout) gets the exception, the script line it came from, how the task's own tests call that function, and their fixtures; if that was the last attempt, it gets **one** extra attempt (3 in all, never more). CONFIRMED is unchanged: every run above still has to come out right. Each check record says which writer made it (`"writer": "v2"`). Switched off, every prompt and record is byte-identical to the writer above (tested).

Why v2 is on: a stop rule we overrode, and say so. Before its re-measure (83 items fixed before any model call) we committed one rule: if any of 46 correct fixes written by other teams became a counted finding, v2 stays off. v2 caught the one miss it was built for, left no item without a valid check (2 before) and flagged 0 of 30 model-written correct controls (95% upper bound 11.4%). One of the 46 outside fixes became a counted finding (1 of 46, 2.2%; 95% interval 0.4-11.3%, Wilson): the raw false-alarm count. Read by an AI model, that fix handles the issue's example but returns the reported wrong answer on a near variant, a defect the task's tests miss; on that reading the count is 0 of 46 false alarms (upper bound 7.7%). The previous writer had confirmed the same defect a day earlier, and the three-company scope vote, not a v2 rule, made it count. The rule could not tell a false alarm from a real defect, so we switched v2 on on 2026-10-18 and published the rule, both results and the reason on our error-rates page. A person outside the company reads the flagged fix; if that person reads it as correct, the published false-alarm figure becomes 1 of 46. Later stop rules count only flags caused by the change being tested that a named reviewer reads as false alarms.

CONFIRMED (pass, fail, fail, then pass and fail again) makes a **finding** with `confirmed_by: "behavioral"`. Its check script, patch and results are kept under `evidence/`. INVALID after both attempts, and ERROR, leave the candidate **unconfirmed**. Unconfirmed candidates are not counted. A difference found only by reading code, with no behavioral check, is never a finding and never sent.

7. Task verdict (`audit.py`)

- **DROP**: preflight failed. The task is not auditable, or is infrastructure-skipped (section 3).
- **FIX**: at least one finding.
- **KEEP**: auditable, at least one candidate ran, and no finding.
- **UNTESTED**: auditable, but no candidate ran (for example, no gold variants and no model). Records made before UNTESTED existed stored KEEP with zero candidates. The report reads those as UNTESTED (`report._effective_verdict`).

8. After the audit

Scope judge (`judge.py`, `plumbline-grader judge`). This runs on saved evidence only. One model call per behavioral finding labels it IN_SCOPE or OUT_OF_SCOPE. The prompt holds the issue, the reference fix (first 30,000 characters), the accepted patch (30,000), the behavior checked, and the check script (20,000).

- IN_SCOPE: the issue asks for the behavior explicitly, or it follows directly from the issue's own example, reproduction steps or expected result.
- Everything else is OUT_OF_SCOPE. This includes behavior only the reference fix covers, other code paths, generalisations, and regressions in behavior the issue does not discuss.
- The judge is told to choose OUT_OF_SCOPE when unsure.

A finding the judge could not label stays unlabeled and never counts as strict. By default, `judge` then adds the run's in-scope findings to the repository's pattern library (`patterns.py`; `--no-learn` turns this off). That changes what the `patterns` strategy shows in later runs, not this run's numbers.

Skeptic (`skeptic.py`, `report --skeptic`). It re-runs a seeded random 10% of findings (at least one; `--skeptic-fraction`, `--seed`) in fresh containers on the recorded image. A finding reproduces when the grader resolves it, it does not equal the reference, and (for behavioral findings) its check passes on the reference and fails on the candidate. A sandbox failure gives no verdict. The non-reproduction rate is non-reproduced findings over checked findings. The report is held when that rate is above 5% or when nothing could be checked. The skeptic also holds a report if any number in its prose cannot be traced to `summary.json`, or if it uses a banned word (certified, attested, guaranteed, unhackable).

Spot check and panel (`spotcheck.py`, `panel.py`). The spot check draws a seeded sample (default 20, seed 0) of IN_SCOPE behavioral findings, at most one per task, with the issue, the reference fix, the accepted patch, the check script, its three results and the scope reason. The sample is for a person to judge each finding genuine or not. The review panel sends the same items (or the whole packet, unchanged) to Claude, GPT and Gemini separately. A finding is confirmed when at least two of three say GENUINE and rejected when at least two say NOT_GENUINE; otherwise it has no majority. The panel judges genuineness, not scope.

9. Sandbox (`sandbox.py`)

Every run (grader, check, file read) uses a fresh container created from the task image and removed afterwards. The flags are:

```
docker create --network none --cap-drop ALL --security-opt no-new-privileges
  --pids-limit 2048 --ulimit nofile=65536:65536 --memory 4g --cpus 2
  --label plumbline-grader=1
  -e CARGO_NET_OFFLINE=true -e PIP_DISABLE_PIP_VERSION_CHECK=1 -e PIP_NO_INDEX=1
  IMAGE sleep infinity
```

- `--memory` and `--cpus` can be changed per run, and the manifest records them.
- The open-file limit is set explicitly because containerd 2.x lowered the default soft limit to 1024 (since 2026-10-04; see the changelog).
- After start, the code inspects the container's network mode and refuses to run unless it is `none`.

Timeouts:

- the swebench eval script and the rebench test command: `--timeout`, default 1800 seconds;
- a Harbor `test.sh`: 10,800 seconds (fixed);
- check scripts: 900 seconds.

Images. An image is pulled only with `--pull`, with retries. A registry rate limit (HTTP 429) is waited out in 5-minute steps for up to 2 hours instead of being recorded as a skip. `--image-source epoch` uses Epoch AI's ghcr rebuild of the SWE-bench images. SWE-rebench-V2 image names from one distribution are rewritten to the same public image on Docker Hub.

Locally built images. A Harbor task without a published image is built from its Dockerfile with the network on, plus a layer that adds python3 and git if missing and commits a git baseline.

For a task whose own environment allows internet (`allow_internet`), there is one more layer. It applies the reference fix, runs the verifier once with the network on to fill dependency caches, then deletes the tests and the fix and restores the repository from a snapshot taken before the warm-up. Grading itself is always offline.

9.1 Network-on mode (`egress.py`, opt-in, from 2026-10-08)

Added 2026-10-08 for tasks whose official harness enables the network. `audit --network-mode network-on` runs only tasks whose `task.toml` gives the verifier the network (`allow_internet`, a public network mode, or no network setting, Harbor's default); every other task is skipped as sealed-only. Each container joins an internal Docker network with no route out; its only way out is a per-container proxy (Python standard library) that logs every request (HTTPS as CONNECT host:port, HTTP as method and URL) to `OUT/egress/<container>.jsonl`, labelled with the task and run label. A request that ignores the proxy settings fails. The sandbox refuses to start if the container's network is not that internal network. In this mode only: `PIP_NO_INDEX` and `CARGO_NET_OFFLINE` are not set, and apt is told to stay root for downloads (`APT::Sandbox::User "root"`), since every capability is still dropped. Records carry `network_mode: "network-on"` and an `egress` summary (requests, failures, hosts); `summarize` refuses to pool them with sealed records, the report is titled NETWORK-ON MODE, and the run must use its own directory. Network-on results are reported separately and never counted in sealed totals.

9.2 Lab-hosted mode (`providers.py`, `egress_guard.py`, opt-in, from 2026-10-10)

With `PLUMBLINE_PROVIDERS` set to a lab's provider file, every model call (the check writer and scope judge, attackers, scope voters, panel) goes only to the endpoints the file names, with the file's URL, key variable, model and prices. In strict mode (the default) a role the file does not name is refused, every request URL is checked against the file, and an in-process guard blocks and records any connection to another host (loopback included, unless listed); a blocked attempt makes the command exit 5 even if code caught it. Network-on mode and network-on image warming are refused. A plain-words description for a lab's security reviewer is available on request.

One model family. When the file allows one family, the method changes in one place: the scope vote's two other votes are independent repeat calls to the same family (`lab#2`, `lab#3`) instead of two other companies, and the attackers are that family alone. Every finding still needs the behavioral check (section 6). Each run writes `model_independence.json` and the report states which families attacked and

voted. When one family did either, the report says so and gives our measured cost: on 27 tasks from one public coding task set, Claude alone found 12 strict tasks, OpenAI alone 13, both 15; a third company added 0 of 12 possible. A one-family count is a lower bound.

10. Numbers in reports (`report.py`, `client_report.py`, `union.py`)

Denominators.

- **Infrastructure-skipped** tasks are excluded from every count except the sample size. `audit --resume` re-runs them.
- **Unauditable** tasks (reference fails, or empty patch passes) are excluded from the hackable shares and listed with their reasons.
- **Auditable** tasks are the rest.
- A task is **tested** by a source when at least one candidate from that source ran (`candidates_run_by_source`). For the combined and strict rows, it is tested when any candidate ran.

Shares.

- *Share of tested* is hackable tasks over tested tasks: the source's rate.
- *Share of auditable* is hackable tasks over all auditable tasks. It is a lower bound, because untested tasks count as not hackable.

Broad and strict.

- **Broad** (the combined row) counts a task as hackable when it has any finding.
- **Strict** counts a task only when at least one finding is labeled IN_SCOPE by the scope judge.
- Rows are also given per source: gold variants and model-written.

The strict share of tested tasks is the headline figure in client reports.

Intervals. Client reports give a 95% Wilson score interval (z = 1.96) for strict hackable over strict tested (`client_report.wilson`). `union.py` uses the same interval for the union of several searches. The certification check uses the Wilson upper bound with z = 1.959964 (`verified.wilson_upper`). All shares are floors: they count only the holes this search found.

Several searches (`union.py`). This counts a task as strict when any run found an in-scope finding. It reports the discovery curve, and gives a Chao2 estimate of the total, capped at the auditable tasks, separately from the confirmed count. It counts only tasks auditable in every run, and flags runs with unjudged findings. `union --by-search` does the same within one run, counting each attacker:strategy search as a separate search; this is the estimate a Plumbline Verified certificate reports (Plumbline Verified criteria v1.0, on the website). Searches inside one run share tasks, checks and judge, so they are less independent than separate runs and the estimate is lower.

Method version. `summary.json` stores the method version of the task records (`method_version`; several versions are listed when a run mixes them), and also the version of the code that rendered the report (`method_version_of_report_code`). `report.md` and the client report print the records' version.

11. Evidence and replay

- Each finding keeps its patch (`*.passed.diff`), its check script (`*.check.py`), the grader log's SHA-256 and the patch's SHA-256.
- `manifest.json` (`manifest.py`, from 0.2.0 on) records the run's code, host, Docker and containerd versions, sandbox flags, models and prompt hashes, and each image's id and digests.
- `plumbline-grader bundle` (`bundle.py`) packages every in-scope finding with its task definition, a `replay.sh` and a SHA-256 list of every file.
- `plumbline-grader replay` re-runs one finding through the same `Runner.evaluate` and `Runner.run_script` code, with the recorded sandbox flags, and compares each result with the record.

Proposal for 0.4.0 (not adopted; written 2026-10-10)

This is a proposal. No default has changed, and every run still uses 0.3.0 as written above.

Measured first. We asked whether the wrong-fix search should aim at the requirements that coverage marks as untested (our internal coverage tool). We tested that on 249 recorded tasks from 3 public coding task sets we had already audited, using the findings those audits proved (132 of the 249 tasks had a strict finding).

- A flagged task had a strict finding 45% of the time (Wilson 95% 34–57%). An unflagged task had one 56% of the time (49–63%).
- A flagged requirement was hit by a strict finding 14% of the time (9–22%). A requirement the tests do mention was hit 48% of the time (41–55%).
- Recall was 23% at the task level and 16% at the requirement level.
- At the largest run's measured cost per task, searching flagged tasks first gives about 1.3 finding tasks per dollar, against 1.7 for the rest.

Proposed for 0.4.0:

1. Keep the search order as it is. Do not aim the `requirements` strategy at uncovered requirements first on code-patch or function-level tasks: on these data it would find fewer findings per dollar.
2. Report coverage holes apart from findings, as a category detector (section 3.1). Never use them as a search priority.
3. Pre-register one test of the opposite ordering before adopting it. The test: on a fresh seeded sample, run the `requirements` strategy on requirements the tests mention first, against the current order, at the same cap, and compare strict findings per dollar. The gap above came from the same data that suggested it, so it does not count as evidence for itself.
4. Repeat the coverage measurement on Harbor script tasks once a script-task audit has proved findings. Coverage was built for those tasks, and this test could not cover them.
5. Add Plumbline Scan PS-12 (`loose_match`, scan 0.3.0) to preflight as a category detector. A short expected value checked by substring is a known false-pass shape: it caught all 6 proved findings on one agent-environment benchmark, with no other flags there. Its hits would be reported apart from findings and would never count as one without the full proof bar.

Changelog

These are changes that can move a number, reconstructed from `git log -- src/plumbline_grader`. Dates are author dates. Every entry up to and including 2026-10-04 `c74802d` ran under the label 0.1.0.

DATE	COMMIT	CHANGE	WHAT IT CAN MOVE
2026-09-30	`082dd76`	First audit pipeline: gold variants, preflight, KEEP/FIX/DROP, skeptic	baseline (0.1.0)
2026-09-30	`1ec0654`	Code-hash equivalence: a finding must differ from the reference as parsed code (docstrings, comments and formatting ignored); UNTESTED verdict; a sandbox failure on a skeptic re-run is no verdict	fewer findings (docstring-only omissions no longer count); KEEP tasks with no candidates become UNTESTED and leave the tested denominator
2026-09-30	`856338a`	Model-written candidates and the behavioral check (pass on reference, fail on base, fail on candidate; one retry)	more candidates per task; findings gated by behavior
2026-09-30	`c55338c`	Registry 429s waited out; `--resume` retries infrastructure skips	fewer infrastructure skips
2026-09-30	`abcbb34`	`--image-source epoch`; each record keeps its image	can change preflight outcomes (a task can be unauditable on the mirror)
2026-09-30	`d86177d`	Scope judge and the strict share	new strict figures
2026-10-01	`544ad02`	Check rules tightened (assert only on the issue's behavior, deterministic, issue's own inputs, `uses_synthetic_state`); scope rule: regressions in behavior the issue does not discuss are OUT_OF_SCOPE	fewer findings and fewer IN_SCOPE labels
2026-10-01	`86b6ab3`, `7e9e4d6`	Harbor tasks graded by their own `test.sh`; separate-verifier mode	new benchmarks graded
2026-10-02	`594246b`	Red team: 64,000-token output, salvage of truncated replies, solution files with test-like names no longer rejected	more model candidates on large tasks
2026-10-02	`9850edc`, `7e4e60f`	Harbor tasks built from their Dockerfile; SWE-rebench-V2 grading; reverse-solve reference fixes inverted; SWE-rebench trees reset before each run	new benchmarks; preflight outcomes
2026-10-02	`cc9ec8b`	Red team reads files up to 8 MB and sees an excerpt around the fix	model candidates on tasks with large files
2026-10-02	`7ceb1ed`	Oracle re-run on confirmation (reference and candidate once more, must agree); prose files left out of the code hash	fewer findings (luck-dependent checks, release-note-only differences)
2026-10-02	`3ff9d06`	Separate-verifier reports read after `test.sh` moves them	test statuses for those tasks

DATE	COMMIT	CHANGE	WHAT IT CAN MOVE
2026-10-02	`b763f1e`	Function-level tasks (ClassEval, BigCodeBench)	new benchmarks
2026-10-03	`76c1c72`	Dependency warm-up layer for tasks whose grader has internet; `CARGO_NET_OFFLINE=true`	fewer unauditable tasks (Java, Rust, Clojure)
2026-10-03	`f2d944b`	Model candidates may create files the reference fix creates	more model candidates on feature tasks
2026-10-03	`9838e91`	SWE-rebench-V2 image names from one distribution rewritten to Docker Hub	which image graded
2026-10-04	`9d6d622`	Container open-file limit set to 65536 (containerd 2.x default soft limit 1024 made one task's grader abort)	preflight and grading on hosts with containerd 2.x
2026-10-04	`bb54e8f`	Attack strategies `requirements` and `patterns`; attackers `openai` and `gemini`; pattern library seeded (22 patterns)	more findings when enabled; defaults unchanged
2026-10-04	`c74802d`	Pattern library rebuilt from audits (28 patterns, 454 findings); `judge` adds each run's findings to it	what the `patterns` strategy shows
2026-10-04	this change	**0.2.0.** Run manifest (git commit, Docker and containerd versions, sandbox settings, model settings, prompt and pattern library hashes, image digests); image id and digests on each task record; summaries state the records' method version; evidence bundle and replay	no verdict rule changed; versions are now recorded
2026-10-08	this change	**0.3.0.** Deeper search is the default for every audit with a model: `--strategies requirements,patterns --attackers claude,openai --reask-mismatched` (0.2.0: `random`, `claude`, no re-ask); an attacker without a key or out of credits is skipped and recorded. Category detectors (section 3.1): oracle stability (reference k=3 times), answer contamination, verifier trusts agent files; reported apart from findings. Script-reference Harbor tasks graded by their `solve.sh`; inline `solve.sh` patches read as the reference fix; a Harbor reward written as a score (`1.0`) counts as full reward (before, only the text `1` did). Same version, later the same day: opt-in network-on mode (section 9.1), records labelled `network_mode`; sealed results unaffected	more model candidates and findings per task; more reference runs per task (k); new category counts. No verdict or finding rule changed
2026-10-10	this change	Lab-hosted mode (section 9.2): model endpoints from the lab's provider file, strict endpoint list, no-egress guard; one-family runs vote with repeat calls and say so in the report	only when `PLUMBLINE_PROVIDERS` is set; standard runs unchanged
2026-10-16	`e3dc9530`	Check writer: fixture `.py` files may be created (reads still rejected); tracked-file guard; repository orientation; retry told the rule and the offending line; every attempt kept	more valid checks on large repositories (recall re-measure of Oct 17: 43 of 50 known wrong fixes counted, from 40; 0 of 150 controls and 0 of 46 outside-written fixes flagged)
2026-10-17	this change	Check writer v2 behind a switch (section 6)	only when switched on

DATE	COMMIT	CHANGE	WHAT IT CAN MOVE
2026-10-18	no code change	Check writer v2 switched on for every audit-server run, overriding the pre-stated stop rule (section 6)	runs started after this use v2 checks; earlier runs keep the previous writer
2026-10-21	this change	**0.3.1.** Version label for the October 16-18 check-writer changes. Records made from Oct 16 until this change carry the label 0.3.0; their run date and manifest commit say which writer ran	no rule changed by the relabel itself

Runs labeled 0.1.0 were made before the 0.2.0 entry, and many were made before some of the earlier entries. For example, an audit run on 2026-09-30 predates the tightened check rules (2026-10-01), the oracle re-run and the prose-file hash (2026-10-02), and the nofile limit (2026-10-04). A re-judged copy of a run applies the newer scope rule, but not the newer check rules. For those, the run date and the commit history say which rules applied; from 0.2.0 on, the manifest records the exact commit.