A finding is only real if someone can act on it

Guide · Watchdog **Version 1.0 - 2026-08-03** Author: Canine Development

Who this is for: the engineer, or the coding agent working on their behalf, who has a Watchdog report and wants to turn it into work. It covers what a finding carries, how the remediation plan is ordered, the calls an agent makes to work it, and what happens when we get one wrong.

A long list of findings is not a plan. Severity tells you how bad each item is on its own; it does not tell you which afternoon of work moves the codebase most. This guide is about the layer between those two things.

The inverse matters too, because it is the easier mistake: **a short list is not automatically a better one.** Fewer findings can mean a cleaner codebase, or it can mean fewer checks resolved on it. That is why a run reports how many applicable dimensions actually returned a result alongside what they found, and marks the rest *not measured* with a reason. A quiet report can then be read as quiet because there was little to say, rather than quiet because little was looked at.

1. What a finding carries

Every observation is a structured record rather than prose you have to parse:

- **A level.** The engine has four: `Info`, `Recommendation`, `Warning`, `Issue`. `Info` is telemetry and is filtered out before findings reach the machine list, so what an agent or a report shows you is Issue, Warning and Recommendation, in that order.
- **A title and a detail** - a headline plus an explanation of what is wrong and why it matters.
- **A location, where the finding has one.** Most findings name a file and a 1-based line. File and line are genuinely optional: architecture, ownership and other repository-level dimensions report against the repository or a subsystem rather than a line, because that is the honest scope of what they measured.
- **A fingerprint** - the stable identity used to claim it, resolve it, and check on the next scan whether it is gone.
- **Provenance** - an optional "detected by" attribution naming the rule or tool behind it, for example a Semgrep rule id.
- **History anchoring.** When a finding lives in git history rather than the working tree - a secret committed months ago - it carries the full commit SHA, and the location points at the blob as of that commit, not whatever occupies that line today.

- **Taxonomy metadata.** Where a detector supplied them, CWE ids and an OWASP category travel with the finding into the report, the SARIF export and the record an agent reads over MCP (`cwe`, `owaspCategory`). Security detectors stamp these; a finding from a non-security dimension carries neither and the fields are absent. We never infer a CWE the detector did not supply.

The point of the structure is that a finding arrives as a work packet rather than a notification.

Here is one, verbatim, from a published survey - the SARIF export for `vnpy/vnpy`, which anyone can download without an account at watchdog.canine.dev/api/oss/vnpy/vnpy/artifacts/report.sarif:

{ "ruleId": "D29", "level": "error", "message": { "text": "High: eval-detected: Detected the use of eval(). eval() can be dangerous if used to evaluate dynamic content. [...] This is a semgrep security-AUDIT rule: it reports that a sensitive construct is present, not that it is exploitable here. Confirm whether this site handles untrusted input or is reachable across a trust boundary - and apply the change where it is; where the construct is required by the platform or protocol it calls into, and carries no untrusted data (a syscall/FFI shim, a build- or debug-gated tool, a fixed local surface), record the review and leave the code as it is." }, "locations": [{ "physicalLocation": { "artifactLocation": { "uri": "vnpy/alpha/dataset/utility.py" }, "region": { "startLine": 252 } } }], "partialFingerprints": { "codehealthFindingId/v1": "d2e677eb255507e3c3c784a5b38c0dc7cbfbb6c9d58dc15834e725b783e32cbd" }, "taxa": [{ "id": "CWE-95", "toolComponent": { "name": "CWE" } }] } ```

Every element of the list above is in that record. The **level** is `error` - SARIF's spelling of `Issue`. The **dimension** is D29, Static Analysis (SAST). The **provenance** is the semgrep rule id `eval-detected`, carried inside the message so you can look the rule up rather than take our word for the call. The **location** is a file and a line. The **fingerprint** is the identity an agent claims, resolves and re-checks against. The **taxonomy** is `CWE-95`, Eval Injection - stamped by the detector on this finding, not inferred from the dimension.

The detail is also worth reading closely, because it does the thing a raw scanner result cannot: it states that this is an *audit* rule, which reports the presence of a construct rather than an exploitable path, and it names the two outcomes - fix it where untrusted input reaches it, or record the review where it cannot. That is what turns a hit into a decision.

2. From a list of findings to "what to fix next"

The remediation plan is a separate object from the findings list, and it is ordered differently.

Each plan row carries a **priority**, an **estimated point gain** (the expected movement in the aggregate score, which is an estimate until the next scan settles it), and a coarse **effort** rating. The plan is ordered by **impact divided by effort**, so the top row is the most score for the least work.

Two refinements are worth knowing, because they explain orderings that would otherwise look wrong:

- **Change frequency breaks ties.** A change-frequency weight multiplies the impact-over-effort key, so of two similar items the one in code your team edits every week ranks above the one in code nobody has touched in a year. It reorders neighbours; it never overrules impact, and it never touches a score.
- **Some rows have no estimable gain.** Work that turns a measurement on - adding the artefact a dimension needs before it can assess anything - is marked as unlocking rather than carrying a point estimate, because the gain genuinely is not knowable until the thing is measurable.

Rows that resolve the same underlying remediation collapse into one, so a single root cause appears once rather than as one line per instance.

One thing to be precise about, because it is easy to state as a rule and it is not one: a low-scoring dimension is *usually* where the headroom is, so it *often* rises to the top. But the ordering is impact over effort, not dimension score. A 2/10 dimension needing a fortnight of work can rank below a cheap fix in a dimension already at 9/10. The exception is a run whose analyzer did not produce a plan: there the audit falls back to ordering by lowest score first, and says so.

3. Working findings with an agent

Watchdog exposes findings to coding agents over MCP at `watchdog.canine.dev/mcp`, authenticated by a per-repository key.

Before anything works: mint a key on the repository's Settings tab, under Automation - the page headed Agent keys - then configure it in your MCP client like any other server. The key is scoped to one repository, and separately to a set of write capabilities: `remediate`, `dispute`, `reportgap`, `rescan`. Reads are not capability-gated, but a key's read scope can be narrowed to particular dimensions, and findings outside it are not returned at all rather than returned and hidden. A call outside the key's capabilities is refused in the response body with the capability named, for example `This key's scope does not grant 'dispute'.` Disputes are additionally capped at 20 per hour per key.

A session then runs:

1. **`audit`** returns the grade, the lens scores and the ranked plan in one call, with each row naming its dimension, current score, estimated point gain, effort and open-instance count. It also tells you whether the plan is impact-ranked or on the lowest-score-first fallback.
2. **`next_task`** hands back the highest-leverage item fully briefed: what the dimension measures, what to do, and the open instances with their fingerprints and locations. `get_task` does the same for a dimension you choose; `list_findings` and `get_finding` browse.
3. **`claim_finding`** takes a short lease so two agents do not fight over one instance.
4. You edit the files.

5. `resolve_finding` marks an instance fixed - provisionally.

Large tasks can carry many instances, and an agent still needs repository access to read the code around them; the payload names the work, it does not replace the working tree.

4. "Resolved" is a claim, not a verdict

`resolve_finding` records that you believe you fixed something. The arbiter is the next scan.

Run `request_rescan`, and if the finding's fingerprint is no longer produced, it is closed; if it is produced again, it reopens. There is deliberately no way to assert a fix and have Watchdog take your word for it.

Three honest caveats:

- Rescan is owner-gated. It draws on the repository's scan budget, so an owner can leave it off, and the call then refuses rather than silently doing nothing.
- A fingerprint that stops being produced is evidence the prior finding was not reproduced. That is what a rescan establishes - it is not a proof that the underlying weakness is impossible.
- A scan that fails or degrades verifies nothing either way. The state stays as it was rather than being read as a pass.

A closed ticket is a promise. A rescan is evidence.

5. When we are wrong

Three routes, each for a different kind of mistake:

- `dispute_finding` flags a *scored* finding as a false positive, with a required reason, and lands for human triage. A person decides, and declining is a normal outcome rather than a failure - bring concrete evidence rather than disagreement. If a maintainer already declined a dispute on that finding, re-disputing is refused and their reasoning comes back with it. That means the finding stands as reviewed and upheld; it is a human decision about the finding, not a proof that the code is wrong, and the escalation path from there is a person rather than another dispute.
- `flag_advisory` is the route for the LLM-judged findings that do not score. They use a different route *because* they do not move your number, not because they cannot be mistaken - an advisory read can be wrong, irrelevant or unhelpful like anything else. A maintainer reviews what you send, and that review is what feeds detector and prompt improvements. There is nothing to re-scan, because nothing scored.
- `report_detector_gap` covers the opposite failure: something real we should have caught and did not. It does not change your score, and we act on it when it is corroborated across repositories or comes with a deterministic reproduction.

Watchdog's design target is below 5% noise, on a deliberately broad definition: a finding counts as noise if it is a false positive, states opinion as fact, duplicates another finding, or is irrelevant to the shape of the code. That is a bar we hold ourselves to, not a measured result - and it is not uniform. Each language is calibrated separately, so some sit well below the bar today and others well above it; the ones added most recently have had the least of that work. The measurement protocol and judge prompt are public at github.com/CanineCC/watchdog-benchmark. These three routes are the main ways to tell us a finding is wrong; they are not a claim to have enumerated every way the product can be.

6. The scope around all of that

- **The plan is produced from Watchdog's own reading of your code**, so the ranking works on a repository with an empty pipeline. It complements CI, linters and scanners rather than replacing them. Some dimensions do need an artefact to exist before they can assess it - coverage needs a coverage report - and those report *not measured* with the reason rather than guessing.
- **On security it reads the code, history, configuration and manifests.** Outside-in penetration testing and DAST against a running deployment are a separate discipline.
- **On compliance it produces evidence, not compliance.** Each run records the commit, timestamps, rubric version and analyzer image that identify it, and a result packaged for sharing is content-hashed and Ed25519-signed so a recipient can confirm it came from us unaltered. That can support an audit. It does not make anyone compliant, and no tool that emits a report can.
- **It scores code, not people.** No developer leaderboard, no velocity metric, no DORA scoring. The history-derived dimensions describe the codebase and where its knowledge sits.

7. Try it on one dimension

Work one item end to end rather than reading a report top to bottom. Point an agent at the repository with its key, call `audit`, then `next_task`, and fix just the top item. Edit the files it names at the lines it names, `resolve_finding` each instance, then `request_rescan` if your owner has enabled it.

Then read what came back. The finding is gone, or it reopened because something was missed, or the scan could not verify it. The headline may or may not move on one item - a single instance in a dimension with many is often invisible at the score level, and that is not a failure. What you have closed is the loop: located, ranked, fixed, verified.

Learn more / verify

- **A finding and a ranked plan as they actually render:** open any project in the measured open-source corpus at cai.canine.dev/surveys - by language, then by project, each with its full report.
- **The agent tool surface** - `audit`, `next_task`, `dispute_finding`, `request_rescan` and the rest - is documented at watchdog.canine.dev/agents, with the live MCP server at `watchdog.canine.dev/mcp`.
- **Noise as a discipline:** the measurement protocol at github.com/CanineCC/watchdog-benchmark, and the scoring methodology.
- **Language coverage:** 13 languages as published on 3 August 2026, TypeScript analysed deeply and JavaScript structurally. The catalogue is live, so read the current one at watchdog.canine.dev/api/public/language-support.