# Code rots even when no one commits

Guide · Watchdog **Version 1.0 - 2026-08-04** Author: Canine Development

**Who this is for:** engineering leads, platform engineers and repository administrators who decide how often a codebase is re-examined. It covers the two schedules Watchdog runs on, what each one checks, how to configure them, and how to tell a clean result from an unmeasured one.

A repository that has not changed in three months is not the same repository it was three months ago. The code is byte-for-byte identical, but the world around it moved: a dependency you pinned picked up an advisory, a framework you build on reached the end of its support window, the one engineer who understood a module went quiet. That gap - the risk that accrues between commits - is what a schedule is for.

---

## 1. Risk arrives without a diff

Most checks in a pipeline fire on a commit: CI runs on push, the linter runs on save, the scanner runs in the pull request. That is the right design for catching what you introduce. It also means that a workflow configured to trigger only on code changes produces no signal at all while a repository sits still, and several kinds of risk arrive with no code change behind them.

- **A new advisory lands against a version you already shipped.** The version string in your lockfile did not move. The advisory that now describes it did. Yesterday's clean pin is today's known-vulnerable one.
- **A credential is still in the tree, or still in the history.** A secret committed last quarter stays reachable in every clone that holds that history, whether or not anyone has touched the file since.
- **A maintainer goes quiet.** The person who held the working knowledge of a subsystem stopped committing. No file changed, but the evidence that anyone still knows those files decayed with the calendar.

A fourth kind of calendar risk sits outside what any repository scan can see: a runtime or framework reaching end of support. Watchdog reports deprecated and vulnerable dependencies, and it does not track vendor support lifecycles - that one belongs on your own migration calendar. It is worth naming here because it moves on exactly the same clock as the three above, and a commit-triggered pipeline is equally blind to it.

---

## 2. Two schedules: the survey and the watch

Watchdog runs on two rhythms.

| | The survey | The daily security watch |
|---|---|---|
| What it does | The full analysis: 127 catalogue entries across ten lenses, folded into one Codebase Assurance Index | The Security & Compliance lens only, over the same checkout |
| How often | At the end of each of your team's sprints; a repository with no team of its own sits on a default 14-day calendar | Once a day |
| What comes out | A scored survey, a trend point, and the four report views | Security findings only, with no effect on the score or the trend |
| Chief exclusion | - | Everything outside the Security & Compliance lens, including dependency hygiene, licence compliance and the git-history dimensions |

The survey is the deep read: it is what produces a score, and its cadence is set where the work is planned rather than on an arbitrary interval. The watch exists to shorten the interval on the fastest-moving risk, so that a new advisory against a package you already ship does not wait for the next survey.

Three separate things decide whether a repository gets a watch on a given day, and they are worth keeping apart:

1. **Included in the package.** Every Watchdog package carries the daily security watch. The current module list per package is on the pricing page.
2. **Enabled for the repository.** It is on by default for every tracked repository and can be switched off in that repository's settings.
3. **Scheduled that day.** A full survey is a superset of the watch, so on a day when a survey already runs for that repository the watch is not queued a second time.

The watch is never billed and never consumes scan quota.

---

## 3. What the daily security watch checks

The watch runs the Security & Compliance lens and nothing else. That is a genuine subset of the survey rather than a separate detector set: the same dimensions, the same thresholds, filtered to one lens. Twelve published dimensions sit in that lens, and all of them run:

- **Dependency vulnerabilities**, from three collectors - the .NET package graph (D30), the npm graph (D33) and the OSV scanner (D38), which covers further ecosystems. Each carries the advisory identifier, text and severity into the finding.
- **Secrets in history and in the tree** (D28). Both passes run: the full commit history, which catches a credential that was committed and later removed but survives in every clone, and the current working tree, which catches live plaintext in tracked configuration.
- **Static analysis** (D29), **infrastructure and container configuration** (D31), and **personal-data handling** (D32).
- **Supply-chain provenance and signing** (D36) and **vulnerability-disclosure policy** (D37).
- **Runtime confinement as declared in your configuration**: network egress (D40), kernel and syscall confinement (D41), and runtime threat enforcement (D42).

Two dimensions readers often expect in that set run on the survey's cadence instead, because they sit in the Readiness lens rather than Security & Compliance: **dependency hygiene (D12)**, which reports outdated and deprecated packages, and **licence compliance (D14)**. Outdated and deprecated reporting also has an ecosystem limit worth knowing before you rely on it: it reads the NuGet package graph, and there is no equivalent implementation for other package managers yet. Vulnerability detection is broader, because OSV covers further ecosystems and npm has its own collector.

Where a collector cannot read an ecosystem at all, the dimension **abstains**: it records that it could not measure, with the reason, and the lens weights re-balance around it. That is the behaviour that matters when you are relying on silence, and it is why §5 asks you to read the reason rather than the absence.

- D10 Test Quality — ~4284 lines of test source are present (.py) but the test-quality collector reads C# only, so skipped/assertion-free tests couldn't be counted. Not scored — this is a gap in the analyzer, not a finding about this repository.
- D11 Test Reliability — Test reliability not included
- D12 Dependency Hygiene — Dependency hygiene not measured — dependency manifest found but not parsed for hygiene
- D14 License Compliance — Not scored — this repository's package manifest is not parsed for licence data yet. A gap in the analyzer's language coverage, NOT a finding that the repository's licenses are compliant (a Python pyproject.toml/requirements.txt (pip/uv/Poetry)), which this pass does not parse yet — so this dimension asserts nothing about this repository's licensing in either direction.
- D17 Explicit Debt — explicit-debt markers are read through a C# workspace today, so they were not read for this repository's language — this asserts nothing about how many markers the code carries. Not scored — this is a gap in the analyzer, not a finding about this repository
- D18 Solution Shape — D18 scores the shape of a .NET solution; this repository has no .NET solution or project files, so the dimension does not apply.
- D20 ADR Quality — N/A — ADRs are expected on deployable products with a user-facing host, not consumed libraries; no ADR log is required here.

Abstentions as a published report prints them, from the survey of psf/requests. Each names the dimension, states that it was not measured, and gives the reason - a Python repository whose dependency manifest was not parsed for hygiene, a licence check with no parser for that manifest yet. Several say outright that this is a gap in the analyzer rather than a finding about the repository. None of them is a pass.

The boundary around all of this: a dedicated software-composition-analysis product maintains its own vulnerability database and traces reachability through the transitive graph. Watchdog reads the dependencies your project resolves and matches them against published advisory data. If your obligations require a composition-analysis tool of record, keep it; the watch is an always-on early-warning layer beside it. And this is source-side analysis throughout - running the application, probing a live endpoint and penetration testing are a separate discipline.

## 4. The two dimensions mined from your history

The least-measured kind of rot is the human kind, and two of the four git-history dimensions address it. Both exist for knowledge preservation: they identify where understanding is concentrated so it can be spread while the people who hold it are still available.

Both are built on the same decay model. An author's evidenced knowledge of a file fades exponentially with a six-month half-life, so a contribution two years old counts for about 6% of a fresh one. The model reads the commit log only - author, date, changed paths.

- **Bus Factor (D16)** flags files where one author holds 90% or more of the living knowledge.
- **Knowledge Freshness (D34)** flags the orthogonal case: files whose total living knowledge has fallen below the model's floor, which is roughly what one focused commit is worth after a year. In practice that means no one has made a substantial change to the file within the decay window.

**Bus Factor findings name people.** The dimension emits one finding per sole owner, titled "Off-boarding risk" followed by the identity in your commit log, ordered by how much single-owned code that person holds. Identities appear on the confidential version of a report, which is served to people with manage authority on the repository; every other reader, including anyone reading a public report, gets the anonymised version.

Naming is what makes the finding actionable - concentrated knowledge cannot be spread without knowing whose head it is in - and the measurement stops there. No per-person score is computed or stored, there is no ranking of people across repositories, and machine accounts are filtered out. On a one- or two-person team single ownership is the ambient state rather than a list of risks. The question these answer is where knowledge is concentrated and who should be paired before their next holiday.

Both signals move with no new commit to the file. The half-life keeps running, so a subsystem that read as healthy shared ownership a year ago can drift toward single ownership because the people who knew it stopped touching it. Two floors keep that honest: a file first committed within the last month is too new to be orphaned, and a repository under active development - 30 or more commits in the last 90 days - can never read as dormant.

## 5. Setting it up

1. **Put the repository on the team that plans its work.** Survey timing comes from the team calendar: a repository is surveyed after each of its team's sprints closes. A repository you have not placed on a team of your own sits on the organisation's default 14-day calendar, so it is still surveyed. Pausing a repository, which holds every survey for it, is on the repository's Settings tab.

2. **Set the sprint length from how fast the code moves.** A codebase in heavy flux warrants a shorter sprint; a stable service in maintenance warrants a longer one, but not never - that is what §1 and §4 are about. The useful ceiling is the point at which you would no longer trust the previous survey to describe the current code.
3. **Leave the watch on, and route it somewhere people read.** Notification rules live on the team page. A rule sends one trigger to one destination - Slack, Microsoft Teams, Discord, a raw webhook or an email address - and the security-watch rule fires only when a new critical finding appears, so a quiet day stays quiet.
4. **Learn the three results.** A dimension that scored is a measurement. A dimension that abstained records the reason it could not measure, and is not a pass. A failed run is neither, and says so. Reading an abstention as a clean bill is the one mistake this whole design is built to prevent.
5. **Treat the history dimensions as planning input.** Bus Factor and Knowledge Freshness point at where to spread knowledge before it is lost: a pairing session, a written walkthrough, a documented handover. Used as a performance measure they would teach people to game the commit log instead.
6. **Keep the tools you already run.** The watch reads your dependencies and your history directly, so it needs nothing else installed and depends on nothing else being present. CI, linters, scanners and a composition-analysis tool of record all keep their jobs; the survey is the periodic whole-codebase layer beside them.

---

## 6. What good looks like

On the daily watch:

- A newly published advisory against a package you shipped last quarter appears as a finding on the next watch, and reaches the team through a notification rule rather than by someone remembering to look.
- A credential in the tree or in the history is surfaced on a schedule rather than at the next deep read.

On the survey's cadence:

- A package that picks up a deprecation flag is read while there is still runway to plan the upgrade, which is one input into choosing the interval.
- A subsystem drifting toward single ownership is on someone's plate before the owner's last week, not after the outage.

And on the configuration itself, all of which you can confirm on the repository and team pages: the watch is enabled, the last run completed, its notification rule points somewhere the team reads, the survey cadence has an owner, and abstained dimensions have been read as gaps rather than passes.

None of that required a commit. The risk moved, so the view of it moved too.

---

## Learn more and verify

- **Surveys, the daily watch and what each package includes:** watchdog.canine.dev and the pricing page.
- **Every dimension named here** - D12, D14, D16, D28 to D42 - with its lens, evaluator and rubric version, in the published catalogue at cai.canine.dev/dimensions. How the lenses fold into one index: cai.canine.dev/spec and the methodology page.
- **Noise and accuracy.** 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.