Which gate to add first

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

Who this is for: the person who decides which automated check the pipeline gets next, and the person who edits the pipeline. Both roles are involved: the ranking is a decision, and closing the top item is a change to CI configuration.

Advice about which code-analysis tool to adopt next usually arrives from someone who sells one, and the category a vendor sells tends to come first in their ordering. This guide answers the ordering question from your repository instead: run one survey, read which classes of automated check your pipeline actually enforces, and close the gap that your own code measures worst on.

---

1. The five classes Gate coverage evaluates

Watchdog's Gate coverage feature reads your pipeline for five classes of automated check. Each is paired with the dimension that measures how your repository is doing on that concern without a gate, and that pairing is what turns a checklist into a ranking.

Class	What the gate enforces	Ranked on
Secret scanning	The build fails when credentials, tokens or keys are detected	**D13** Secret Scanning
Dependency vulnerability scanning	The build fails on a dependency matching a published advisory	**D12** Dependency Hygiene
Licence compliance checking	The build fails on a dependency licence you have not accepted	**D14** Licence Compliance
Test-coverage gating	The build fails below a coverage threshold	**D8** Code Coverage
Debt and suppression ratchets	The build fails when suppressions, TODOs or warnings grow	**D17** Explicit Debt

Two things this table is not. It is not a claim that these five are the only automated checks a pipeline can have - they are the five this feature evaluates. And a gate enforces a rule at a moment in the workflow rather than guaranteeing an outcome: a push-triggered secret scanner fails a build after the credential has reached the branch, which is why rotation is still the first response to a real one.

Each class has a mature market and several credible open-source options, and the right answer differs by ecosystem. The right secret scanner for a Go monorepo is not the right one for a .NET solution, and you know your ecosystem better than any comparison table.

---

2. Why the order is a measurement

A gate is worth adding in proportion to how badly the thing it guards is currently going. That is a property of your code rather than of the tool.

Two teams with byte-identical pipelines can have opposite correct answers. A repository with clean dependencies and three hard-coded credentials in its history should close the secret-scanning gap first. A repository with immaculate secret hygiene and dependency hygiene scoring 3 out of 10 should close the dependency gap first. A generic checklist gives both teams the same advice, and is therefore wrong for at least one of them.

Watchdog already measures each of those five concerns, whether or not you gate on them. So the ranking is:

> The classes your pipeline does not enforce, ordered by how badly this repository measures on the concern each one covers.

Worst measurement first. A concern the survey could not measure sorts last rather than first, because a rank should not be awarded on the strength of a number that does not exist. Where two gaps measure equally badly, the tie goes to the one whose failure mode is least recoverable - a leaked credential cannot be un-leaked, and debt can be paid down later.

That is a survey result rather than a checklist, and it moves when your code moves. It is still a model: the pairing in §1 and the choice to rank on measurement rather than on effort or business impact are both editorial decisions, published here so you can disagree with them knowingly.

---

3. Run the survey and read the card

The first full report on a repository is free and needs no card. Nothing is installed in your repository or your pipeline: connect a repository through the read-only GitHub App, or add a public repository by owner and name, at watchdog.canine.dev.

When the survey finishes, the repository's Overview carries a **Gate coverage** card showing how many of the five classes your pipeline fails the build on, and naming the one to close first. Its colour is worth understanding:

- **Green** - every class is enforced. This feature has nothing further to recommend.
- **Amber** - classes are unenforced, and the concerns they cover currently measure at or above the midpoint of the scale. Worth closing so they stay that way.
- **Red** - at least one unenforced class covers a concern this repository currently scores below 5 out of 10 on.

The card is deliberately **not** coloured by how many classes are missing. Four gaps on a healthy repository is a smaller problem than one gap on a repository leaking credentials, and counting would say the opposite.

The card deep-links to the full ranked list under Reports. That list opens by naming the class to add first, then gives, for each one, the state of your pipeline, a sentence putting the rank in your repository's terms, and the score behind it - **out of 10, where 10 is best**, labelled with what it measures. For example: *Secret scanning is not in your pipeline. This survey scores credentials reaching the repository at 3.1/10 on the current commit.*

A repository with no CI pipeline at all is reported as its own case rather than as five missing gates: there is nowhere to add one yet, and that is the first thing to fix.

---

4. Reading the three states

Enforced. A real invocation in your pipeline that can fail the build. Nothing further for this feature to recommend on that class - which is narrower than saying the check is well configured, since §6 explains what is not assessed.

Advisory only. The check runs, reports, and cannot fail anything: a step marked `continue-on-error`, or a tool invoked with a flag that swallows its exit code. This counts as unenforced. It is worth particular attention because the tool is present, the finding is generated, and everyone assumes something is being guarded. Nothing is being stopped. That is a reason to look at it early, and it is separate from technical severity: an absent gate on a badly failing concern can still be the more urgent of the two, and that is what the ranking is for.

Not in pipeline. No invocation of that class in any CI configuration Watchdog can read.

Detection is deliberately strict. Comments are stripped before matching, a tool named only in a step's display name earns nothing, an install or setup line is not an invocation, a tool name has to appear as a command or a `uses:` reference rather than as an argument, and a setting that switches a check off is not a gate. A tool whose failure is suppressed is downgraded to advisory rather than credited as a gate.

The error to expect from all that is being told a gate is missing when you believe it is there. Crediting a gate that is not really enforcing is the one this works hardest to avoid, because it is the mistake that would cost you something, and it is why §5 asks you to check the top item before you spend anything.

---

5. Working the list

Work the ranked list from the top. That is the whole method: the order already accounts for what your code is doing, so rank one is the recommendation.

Check the top item before you spend anything. If the class already runs somewhere Watchdog cannot see - a CI system outside version control, an organisation-level policy - the ranking is wrong about your pipeline, and the next item down is your real rank one. If the top item is Advisory only, closing it is usually configuration rather than procurement: removing the flag that suppresses the failure. That is the cheapest rank you will ever close, and it does not change the order - it just means rank one is nearly free today.

Add or enforce one class, then re-run the survey. The class should move to Enforced once a build-failing invocation is committed and Watchdog recognises it. If it does not move, the invocation is in a form the detector does not read, and the recorded state will say which of the three it saw.

Expect the order to change. Close a coverage gap while dependency hygiene drifts, and dependency scanning will rise. That is the ranking following the code, not the ranking being unstable.

When all five read enforced, this feature has nothing left to recommend. That is the end of this particular list rather than the end of useful investment in your pipeline: it says nothing about how well any of the five is configured, and nothing about controls outside the five.

---

6. What this does not tell you

Naming the class is where the recommendation stops. Watchdog does not name a product, compare vendors or rank the options inside a class, and it receives nothing from whichever you choose. That constraint is structural rather than a courtesy: Watchdog scores repositories that run these tools, and a recommendation from the party that then grades your use of it is worth less for that reason. Note what the narrow claim is: no compensation depends on which third-party tool you pick. Watchdog is not a disinterested party in general - it would rather you ran the survey.

It reads the pipeline **as committed in your repository**: GitHub Actions and the compatible forge layouts, GitLab, Jenkins, Azure DevOps, Bitbucket, Travis, Drone, AppVeyor and Buildkite, plus the scripts those pipelines invoke. A gate that runs in a system outside version control is invisible to it, and will be reported as missing.

And it reads **whether** a class is enforced, not how well the tool behind it is configured. A secret scanner wired to fail the build counts as enforced whether its rule set is thorough or minimal.

6.1 Ask any tool how much of itself ran

One question is worth asking every tool you are sequencing: **how much of what you claim to check actually resolved on my code?**

A low error rate is cheap to achieve by attempting little. A tool that attempts four checks and reports almost nothing is quiet, and quiet reads as accurate. So ask for two numbers together - the share of applicable checks that returned a real result rather than being skipped or inconclusive, and the error rate over that same set. Either alone is hard to interpret.

Watchdog records, on every run, how many applicable dimensions returned a result and how many abstained, and publishes per-language depth rather than one blended figure. The definitions behind both numbers, and Watchdog's own, are on the methodology page.

Watchdog is not one of the five gates and does not become one. It surveys the whole codebase periodically; the gates guard the door on every change, narrowly. Both are worth having, which is why this guide exists.

---

7. Where to go next

- **Run the free survey** and read the Gate coverage card: watchdog.canine.dev
- **The five dimensions the ranking reads** - D8, D12, D13, D14 and D17 - with their lenses and evaluators: cai.canine.dev/dimensions
- **How the survey scores your code**, lens by lens: watchdog.canine.dev/methodology
- **Where the survey sits** relative to the tools you already run: watchdog.canine.dev/where-watchdog-fits
- **The open scoring standard** behind the index: cai.canine.dev/spec