Appendix: Practical Gate Implementation

The Quality Gates chapter defines six tiers (five machine, one human). The Gate Tooling appendix lists specific tools. This appendix covers the part in between: how to actually implement each tier if you are a normal engineering team without a research lab or a dedicated platform team.

The companion repository at github.com/brennhill/Delivery-Gap-Toolkit contains runnable scripts for several recommendations in this appendix, including a delivery metrics calculator, a cost-per-accepted-change tool, and a multi-pass AI code review script.

TIER 0 – STATIC ANALYSIS

Static analysis is the highest-leverage, lowest-effort gate. The goal is to catch entire categories of problems before any human sees the code. Everything in this tier runs in CI and blocks the PR on failure.

PR size limits

Enforce a maximum PR size in CI before anything else. The SmartBear/Cisco study (2,500 reviews, 3.2 million lines of code) found that reviewers detect 70-90% of defects at 200-400 lines, but detection rates collapse beyond 400 lines. PRs over 1,000 lines show 70% lower defect detection. After 60 minutes of continuous review, effectiveness drops sharply regardless of reviewer skill. Set a hard limit at 400 lines. Reject anything larger and require it to be split — if AI generated it, AI should split it. A 1,200-line PR is not a review. It is a rubber stamp waiting to happen.

Linter categories

Most teams start with a code style linter and stop there. Tier 0 requires several categories of linting, each catching a different failure class:

Code style and formatting. ESLint, Prettier, Ruff, gofmt, rustfmt, or equivalent. Eliminates style debates and catches basic syntax issues. The specific tool matters less than the fact that it runs on every PR and blocks on failure.

Type checking. TypeScript strict mode, mypy, pyright, or equivalent. Catches type errors that AI-generated code introduces confidently and silently. AI models produce plausible-looking code that often has subtle type mismatches.

Dead code analysis. Tools like vulture (Python), ts-prune (TypeScript), or built-in compiler warnings. AI-generated code frequently leaves behind unused imports, unreachable branches, and orphaned functions. Dead code is not just clutter — it is surface area for confusion in future AI sessions.

Security linting. Language-specific security scanners: gosec (Go), Bandit (Python), Brakeman (Ruby), ESLint security plugins (JavaScript). These catch common vulnerability patterns — hardcoded credentials, SQL injection, path traversal — at the syntax level.

Semgrep. Deserves its own category. Semgrep runs custom and community rules across any language. Start with the `p/default` and `p/security-audit` rulesets. These cover OWASP patterns out of the box. You can add custom rules later for your domain-specific patterns.

Duplication detection. Duplicated code blocks rose eightfold in 2024 as AI adoption increased (GitClear, 2025). General code quality platforms flag duplication, but dedicated tools like jscpd (150+ languages, MIT) or PMD CPD (JVM, C, Go, Python) catch it faster and enforce hard thresholds in CI. Set a maximum duplication percentage per PR and block on violation. This is not optional — without it, AI tools will generate near-identical blocks across your codebase instead of extracting shared logic.

Code quality platforms. SonarCloud, Codacy, or Snyk Code. These are SaaS tools that connect to your repository and scan every PR for code smells, complexity, duplication, and security vulnerabilities. They provide trend dashboards — useful for answering "is our code quality improving or degrading over time?" Connect your repo; they start scanning. Most offer free tiers for open source or small teams.

Dependency verification. AI-generated code hallucinates package names at significant rates — 5.2% for commercial models, 21.7% for open-source (Spracklen et al., USENIX Security 2025). Tools like sloppy-joe verify that every dependency actually exists on its registry, flag names suspiciously close to popular packages (typosquatting), and enforce your team's canonical package choices. Run in CI before `npm install` or `pip install` — not after.

Secret detection. TruffleHog, detect-secrets, or GitGuardian. Catches credentials, API keys, and tokens before they reach the repository. This is a separate category from security linting because the detection patterns are different — regex and entropy-based, not syntax-based.

When the linter finds 500 issues on day one

If you are adding static analysis to an existing codebase, you will get a wall of failures. Do not try to fix them all before enabling the gate. Instead:

1. Run the linter once and record the current baseline.
2. Configure the gate to block only on *new* issues introduced in a PR, not existing ones.
3. Fix existing issues incrementally, one category at a time, as part of normal work.

Most tools support this baseline approach natively. SonarCloud calls it "new code period." Semgrep supports --baseline-commit. The goal is to stop the bleeding first and clean up the wound later.

TIER 1 – CONTRACT GATES

Contract gates verify that interfaces between services behave as documented. The most common implementation is API snapshot testing.

API snapshot approach

If you have OpenAPI specs, use oasdiff or Optic to diff the spec against the implementation on every PR. Any breaking change — renamed field, changed type, removed endpoint — fails the build.

If you do not have OpenAPI specs, start by recording production responses. Tools like Pact, Dredd, or simple snapshot scripts can capture the shape of your API responses (field names, types, structure) and store them as contract files. Future PRs are validated against these snapshots. Any drift is flagged.

For GraphQL, GraphQL Inspector or Apollo GraphOS Schema Checks serve the same purpose.

The details of setting up contract testing are well-documented online and somewhat specific to your stack. The key principle for this book: a contract check must exist, it must run in CI, and it must block. The specific tool matters less than the enforcement.

What to do when the spec is wrong

Sometimes the contract check fails because the spec is outdated, not because the code is wrong. This is a healthy signal. Update the spec, get it reviewed, and merge. The gate forced you to make the contract change explicit rather than silent. That is the gate working, not the gate being annoying.

TIER 2 – INVARIANT GATES

Invariant gates verify properties that must hold for all inputs, not just the inputs you wrote test cases for.

The Quality Gates chapter provides the core method: ask three questions about any feature. What must never happen twice? What must always be true after this operation completes? What breaks if operations run out of order? The answers define your invariant tests.

Implementation

Property-based testing frameworks generate random inputs and verify that invariants hold across all of them. Hypothesis (Python) and fast-check (JavaScript/TypeScript) are the most mature. A property-based test for an idempotency invariant might look like: "for any valid order, calling the checkout endpoint twice produces exactly one charge." The framework generates hundreds of random orders and verifies the property holds for each.

For teams not ready for property-based testing, standard integration tests that explicitly assert invariants are a practical starting point. The difference between a regular test and an invariant test is intent: a regular test checks "does this return the right value for this input." An invariant test checks "does this property hold for every input, including the ones I did not think of."

Add invariant tests to your CI pipeline the same way you add unit tests. They run on every PR. They block on failure. The specific framework matters less than the discipline of identifying and testing the properties that must never break.

TIER 3 – POLICY GATES

Policy gates encode organizational rules as executable code. Security policies, compliance requirements, permission boundaries — these are not suggestions. They are constraints that must be enforced automatically.

Agent sandboxing is non-negotiable

For any truly agentic workflow — where an AI agent executes code, calls APIs, or modifies files — sandboxing is not optional. It is the minimum viable safety measure. The OpenAI misalignment monitoring data from the Breakdowns chapter showed that agents routinely treat security controls as obstacles to solve. They extract credentials, bypass content scanning, and evade controls through string manipulation. They do this not out of malice but because they do not distinguish between "solve the coding problem" and "solve the security problem that is in the way."

NVIDIA OpenShell is the strongest open-source option for hardened agent sandboxing. It implements default-deny on filesystem, network, process, and inference. Permissions are added via declarative YAML policies — the agent starts with nothing and earns access one policy at a time. Kernel-level enforcement via Landlock, Seccomp, and OPA/Rego means the agent cannot bypass the constraints programmatically.

Docker is the minimum. Run agents in containers with no host mount, no outbound network except allowlisted endpoints, and no environment variable passthrough. This is not hardened — a determined agent can potentially escape a standard container — but it bounds the blast radius significantly compared to running on the host machine.

gVisor sits between Docker and OpenShell. It adds syscall filtering to container isolation, intercepting system calls before they reach the host kernel.

Firecracker microVMs provide the strongest isolation short of a separate physical machine. Used by AWS Lambda and Fly.io. Higher operational overhead, but the blast radius is bounded by hardware virtualization.

Pick the level that matches your risk tier. Risk Tier 1 code (documentation, test scaffolds) can run in Docker. Risk Tier 2 (customer-facing features) should run in gVisor or OpenShell. Risk Tier 3 (billing, auth, compliance) should not have autonomous agent access at all, but if it does, Firecracker or equivalent is the minimum.

Policy scanning in CI

Semgrep custom rules, OPA/Conftest for infrastructure configs, and dependency audit tools (npm audit, pip-audit, Snyk) catch policy violations before merge. CODEOWNERS files on risk Tier 3 directories force the right reviewers onto every change.

TIER 4 – BEHAVIORAL MONITORING

Behavioral monitoring catches problems that no static check, contract test, invariant, or policy gate can see. It detects anomalous patterns in what agents actually do, not what they were supposed to do.

This is the hardest tier to implement. The infrastructure to do this well outside a frontier AI lab is still limited, and honesty about that matters more than pretending otherwise.

What is available today

Tracing platforms. Langfuse (open source), Arize Phoenix (open source), and Langsmith (commercial) capture structured traces of every agent action: tool calls, file accesses, API requests, token usage, session duration. Setting up trace collection is the prerequisite for everything else in this tier.

Basic anomaly detection. Once you have structured logs, you can build simple baselines and alert on deviations. The approach:

1. Collect structured logs for your first 50-100 agent sessions.
2. Compute simple statistics: median file reads per session, median tool calls, median token usage, median session duration, p95 for each.
3. Set alert thresholds at 3x the p95 value. An agent that normally reads 5 files but suddenly reads 50 is worth investigating. A session that normally uses 10,000 tokens but spikes to 100,000 is worth investigating.

4. Track which tools the agent calls. Any tool call the agent has never made before in your baseline period is worth flagging.
5. Tune thresholds after the first month based on your false positive rate. If you are getting more than one alert per day that turns out to be nothing, widen the thresholds.

These are starting defaults, not research-grade detection. They will catch gross anomalies — credential extraction, runaway sessions, scope violations — but they will not catch subtle drift or sophisticated evasion.

LLM-as-judge sampling. Take 5-10% of agent sessions and send the full trace to a separate model with a review prompt: "Given this task description and this execution trace, did the agent stay within its documented scope? Did it access resources beyond what the task required? Did it do anything unexpected?" Flag anything the judge scores as suspicious for human review. Track the judge's agreement rate with human reviewers over time to calibrate trust.

Spotify's Honk system uses this approach and vetoes approximately 25% of sessions. That veto rate is the system working — it means the judge is catching scope creep that deterministic checks miss.

What is not yet available

Production-grade behavioral baselining — the kind that detects a 2% quality drift over weeks or catches an agent whose behavior shifts gradually rather than suddenly — requires infrastructure that most organizations do not have. OpenAI built this for their internal agents using GPT-5.4 Thinking at maximum reasoning effort across tens of millions of interactions. That is not a realistic starting point for most teams.

The honest recommendation: start with trace collection and basic anomaly detection. Add LLM-as-judge sampling when your agent volume justifies it. Watch the tooling ecosystem — this is the tier where the gap between what frontier labs can do and what normal teams can do is largest, and it is closing fast.

The companion repository at **github.com/brennhill/Delivery-Gap-Toolkit** contains a step-by-step agent monitoring guide with worked code examples for trace setup, baseline computation, alert thresholds, and LLM-as-judge prompts. The guide is maintained as the tooling evolves — check there for the latest recommendations.

INSTRUMENTING THE METRICS

The Verification Triangle chapter and the Mandate chapter tell you to instrument four numbers: cost per accepted change, change failure rate, reviewer-minutes per accepted change, and rework rate. This section covers the tools that collect them.

Engineering intelligence platforms

These platforms connect to your Git provider and CI system and compute delivery metrics automatically. They are the fastest path from "we have no data" to "we have a weekly scorecard."

Tool	What it does	Open Source	URL
LinearB	DORA metrics, cycle time, PR analytics, investment allocation. Connects to GitHub/GitLab/Bitbucket	No (free tier available)	https://linearb.io
Sleuth	DORA metrics, deploy tracking, change failure rate, custom metrics. Git + CI + issue tracker integration	No (free tier available)	https://sleuth.io

Tool	What it does	Open Source	URL
Swarmia	Engineering effectiveness metrics, investment balance, working agreements. GitHub/GitLab integration	No (commercial)	https://swarmia.com
Haystack	DORA metrics, sprint analytics, PR cycle time. GitHub/GitLab/Jira integration	No (commercial)	https://usehaystack.io
Faros AI	Engineering intelligence from 50+ data sources. The source of the "98% more PRs, zero throughput gain" data cited in this book	No (community edition available)	https://faros.ai

If you do not want a platform

A spreadsheet and git history are sufficient to start. The Mandate chapter says this explicitly, and it is true.

Merged PR count. `git log --merges --since="4 weeks ago" | wc -l` or pull from your GitHub/GitLab API.

Rework rate. Ask your team each Friday: "Did anyone look at something we shipped and realize we built the wrong thing?" Count the changes where the approach was wrong, the intent was misunderstood, or the design had to be reconsidered — not just bug fixes. Divide by total merged changes. This is manual and imprecise. It is also better than nothing, and it takes five minutes per week.

Reviewer-minutes per accepted change. Most code review platforms (GitHub, GitLab, Gerrit) record review timestamps. Pull the time between "review requested" and "approved" for each merged PR. Divide total review time by accepted changes. If your platform does not expose this, estimate: ask your senior engineers how many hours they spent reviewing this week.

Change failure rate. Count deployments that caused production degradation requiring remediation — rollbacks, hotfixes, or corrective action. Divide by total deployments. This is a DORA core metric; most teams already have tooling for it. If you also find defects that reached production without causing degradation, track those separately as non-degrading escapes — each one is a gating opportunity.

Cost per accepted change. The hardest to compute and the most valuable. Add up: model/API spend (from your vendor billing), infrastructure cost (from your cloud bill), human engineering time — discussion, whiteboarding, spec writing, prompting, context preparation (estimated hours × burdened rate), review time (from the reviewer-minutes metric above × burdened rate), and rework cost (estimated hours on fixes × burdened rate). Divide by accepted changes. The first calculation will be rough. That is fine. Directional honesty matters more than false precision.

THE WEEKLY SCORECARD TEMPLATE

The Mandate chapter requires a weekly review with a pre-read. This is the pre-read. One page, four numbers, one outlier, one control tweak.

Scorecard

Metric	This week	Last week	4-week trend	Target
Cost per accepted change	$	$	↑ ↓ →	$
Rework rate	%	%	↑ ↓ →	< %

Metric	This week	Last week	4-week trend	Target
Reviewer-minutes per accepted change	min	min	↑ ↓ →	< min
Change failure rate	%	%	↑ ↓ →	< %

Fill in the target column with your team's baseline from the first four weeks of measurement. The target is not aspirational — it is the number you are trying to hold or improve. If any metric moves more than 15% in the wrong direction for two consecutive weeks, that is the outlier for the meeting.

Outlier autopsy

Pick the single most interesting PR from the week — the one that was most expensive, most reworked, most surprising, or most instructive. Answer three questions:

1. What happened?
2. Which vertex of the triangle was weak? (Intent clarity, verification quality, or cost?)
3. What one control change would have caught it earlier?

Control tweak

Pick one change to make this week based on the scorecard and the outlier. One. Not three. Examples: "Add a contract check on the payments endpoint." "Require spec link on Tier 2 PRs." "Set up secret detection in CI." Assign it to a named owner. Review whether it was done in next week's meeting.

Meeting format

Fifteen minutes. Three people: EM, tech lead, one rotating IC. Pre-read posted the day before.

- Minutes 0-5: review the scorecard.
- Minutes 5-10: autopsy the outlier.
- Minutes 10-15: pick one control tweak and assign an owner.

If the meeting consistently runs under fifteen minutes, you are doing it right. If it consistently runs over, you are trying to solve too many problems at once. Pick one.