THE NARROW PIPE

A Position Paper on the Emerging Challenges in Developer Infrastructure for Agent-Scale Software Engineering

With Specific Considerations for Mature Monorepo Environments

March 2026

Author: Roger Fleig

Executive Summary

The developer productivity field has spent decades optimizing a pipeline that assumes human-speed code production: build systems, CI/CD, code review, test infrastructure, and merge queues. Every one of those systems was designed around the throughput of humans typing code.

When AI agents drop the cost of producing code by an order of magnitude, the bottleneck does not disappear. It shifts downstream, and every existing constraint in the pipeline becomes the new critical path. This is the classic Theory of Constraints applied to software engineering infrastructure: the constraint was producing code; now it is everything after producing code. Beyond throughput, there is a subtler risk: without deliberate reinforcement, the pipeline does not merely constrict the flow of agent-generated code — it allows institutional quality to leak out over time, as review rigor erodes under volume pressure and agent output becomes the context for future agents.

The goal is no longer "more code." It is ensuring that agentic speed translates into business outcomes without eroding architectural stability. The infrastructure that governs code quality — review, testing, integration, deployment — must be rebuilt for a world where the volume of changes exceeds what human judgment alone can govern. This paper maps the problem space of agentic development infrastructure, identifies the key design tensions, and proposes a framework for structured experimentation grounded in established measurement science (DORA, SPACE, DevEx). It gives particular attention to mature monorepo environments — large, long-lived codebases where coupling is high, institutional knowledge is encoded in developer culture rather than documentation, and integration testing is expensive and often unreliable. These environments amplify every challenge the agentic era introduces.

This is a living document intended to evolve through experimentation and iteration. A note on posture: this is an opinionated position paper. It advances a thesis, selects evidence in service of that thesis, and proposes an agenda. Many of the claims — particularly around higher-autonomy workflows, the economics of verification, and the infrastructure responses proposed — are working hypotheses rather than settled findings. For readability, we have not labeled every inference as such. The experimental framework in Section 4 exists to sort what we believe from what we can demonstrate.

---

1. The Narrow Pipe Thesis

The core observation is simple: the cost of producing code is falling rapidly. The downstream infrastructure that reviews, builds, tests, integrates, and deploys that code was designed for human-speed throughput. When agent-generated code floods this pipeline, the bottlenecks that were tolerable at human scale become critical at agent scale.

This is not a hypothetical future. Teams experimenting with agentic workflows today are already encountering these constraints: CI queues backing up from agent-driven retry loops, code review backlogs from agent-generated PRs, and merge conflicts from concurrent agent work on shared codebases. In mature codebases, once agentic code production becomes cheap enough, **the limiting factors shift disproportionately downstream into verification, coordination, and governance.** The dominant bottleneck will vary by environment, task class, and organizational maturity — but the direction of the shift is consistent.

1.1 What Changes, What Doesn't

Much of the hard-won knowledge from decades of developer productivity work remains relevant. Build optimization, test reliability, dependency management, code health metrics — these do not become obsolete. But they were optimized for a world where the scarce resource was developer time producing code, and the abundant resource was time to review and test. Those ratios are inverting. The experience of having optimized for the old world is precisely what qualifies a team to recognize what needs to change for the new one.

1.2 The Monorepo Amplifier

In a polyrepo world, agents working on different services are naturally isolated. In a monorepo, every change is potentially a global change. The existing tooling around visibility, ownership files, and dependency analysis assumes a human developer who understands the blast radius of their change and self-limits. An agent does not have that intuition. It will happily modify a shared utility function because it is the cleanest fix, not realizing it triggers rebuilds and test runs across the entire organization.

> **The monorepo's greatest strength — atomic cross-cutting changes — becomes a liability when agents can produce those changes trivially and without judgment about whether they should.**

This means the build and test infrastructure, already the most expensive shared resource, faces a demand spike it was not designed for. Imagine every developer has five to ten agents, each iterating on changes and triggering test runs in a loop until tests pass. That iterate-until-green pattern is an unbounded multiplier on CI load.

1.3 The Agent Loop as the Unit of Work

To reason about agent productivity, it helps to define the **Agent Loop** — the functional unit of agentic development — in two layers:

The Micro-Loop: The agent's internal cycle of Reasoning → Action → Observation. Each iteration consumes tokens, time, and potentially infrastructure resources (e.g., a test run). High-performing agent environments minimize the number of micro-loop iterations needed to complete a task by providing rich context, reliable tools, and clear feedback signals. A flaky test that sends the agent down a wrong path wastes many micro-loop iterations. Poor context injection that omits a key dependency wastes more. Every problem described in this paper can be understood as something that either inflates or deflates the micro-loop count.

The Macro-Loop (Human Steering): The roundtrip between an agent and a human engineer — a review comment, an escalation, a course correction. The macro-loop is where human judgment enters the system. A useful metric here is **Autonomy Duration**: how far an agent can progress through meaningful work before requiring human intervention. Longer autonomy duration means fewer macro-loops, which means less human bottleneck. But longer autonomy without adequate guardrails means higher risk of compounding errors.

The Narrow Pipe thesis, restated in these terms: agents create pressure on both loops simultaneously. The micro-loop hammers shared infrastructure — builds, tests, CI — at 10-100x the rate it was designed for. The macro-loop stays human-speed but now faces a volume of agent-produced work that no review process was built to absorb. The pipeline gets squeezed from both directions, and the downstream infrastructure designed for human-speed code production becomes the bottleneck at every stage.

1.4 The Autonomy Spectrum: From Autocomplete to the Dark Factory

A useful taxonomy for the space comes from Dan Shapiro (January 2026), who mapped the progression of AI-assisted development onto five levels — explicitly modeled on the NHTSA's five levels of driving automation. The analogy is intentional: each level shifts who is driving. We use this taxonomy as an analytic convenience for discussion, not as a validated maturity model — the claim that Levels 3 and 4 are where the infrastructure investment matters most is this paper's argument, not something the taxonomy itself establishes.

Level 0 — Manual. Code is written entirely by hand. AI is absent or an afterthought.

Level 1 — Autocomplete. AI suggests completions as you type. The human is still driving; AI is cruise control on a straight road. The workflow is unchanged.

Level 2 — Pair programming. The developer pairs with the AI as a colleague, handing off routine work and getting into flow. This is where the vast majority of "AI-native" developers operate today. Critically, every level from here on feels like the ceiling. It is not.

Level 3 — Code review (Human in the loop). The dynamic flips. AI writes the code; the developer reviews it. The developer is no longer a senior engineer — that is the AI's job. The developer is a manager. This is where Stripe's Minions operates (Appendix A), and it is where the Narrow Pipe thesis bites hardest: the review burden becomes the bottleneck.

Level 4 — Spec-driven development. Engineers write detailed specifications covering what the software should do, how it should behave, and what the acceptance criteria are, then hand them to AI agents. Hours later, they check the results against specs and tests. The human's job shifts from *how* to *what*. StrongDM's "Attractor" system operates here: specifications go in as markdown, agents write the code, other agents test it against holdout scenarios the coding agents never see.

Level 5 — The Dark Factory. Specs go in, software comes out. The human role is defining what to build and why. The how is entirely autonomous. Named after FANUC's 1980s "lights-out" robotics factory where robots built robots with no humans present.

It is worth pausing here to name a confusion that runs through most public discourse on this taxonomy. Levels 1 and 2 make *one developer faster*. That is a real gain, and it is the gain most organizations are measuring and celebrating. But the transition to Level 3 and beyond is not a continuation of the same curve — it is a different curve entirely. At Level 3, a developer stops writing code and starts reviewing it. There is no reason to manage just one agent at a time. The moment the workflow flips from writing to reviewing, the natural unit of work becomes *a portfolio of concurrent agents*, not a single accelerated individual. The productivity model shifts from linear (one human, one thread, faster) to multiplicative (one human, N threads, in parallel). Organizations that frame AI adoption purely as individual-developer productivity improvement will optimize for Level 2 indefinitely and never encounter the infrastructure constraints this paper is about — because they will never stress-test them. Those constraints only emerge when the pipe fills up.

Where This Paper Sits on the Spectrum This paper is primarily concerned with **building the infrastructure for Levels 3 and 4 to work reliably at scale** — particularly in mature monorepo environments where the structural challenges (blast radius, context injection, legacy code, flaky tests) make each level transition harder than in greenfield settings. Level 2 is already happening inside most large engineering organizations. Level 5 may be viable for small teams on greenfield projects, but the coupling, institutional knowledge, and integration complexity of mature monorepos make it structurally premature for those environments.

The practical focus is on making Levels 3 and 4 work reliably at scale — and measuring rigorously enough to know when they do.

Autonomy, controls, and governance An important extension emerging from the community discourse around Shapiro's levels: autonomy alone is not maturity. Increasing what the AI does (autonomy) without proportionally improving the ability to verify what it did (controls) or to manage permissions, audit trails, and accountability (governance) creates risk, not progress. This observation is straightforward but frequently overlooked in the enthusiasm around advancing up the autonomy spectrum.

This framing maps directly to this paper's structure. Section 2 ad-

dresses the problem landscape across all three axes. Section 3 provides the measurement framework to determine whether controls and governance are keeping pace with autonomy. Section 4 proposes experimentation as the methodology for making progress without making reckless bets.

---

2. The Problem Landscape

The challenges of running agents safely and productively decompose into eight interdependent problem areas. Each has its own design tensions, and each represents an area where structured experimentation can produce actionable results. Not all eight are equally central to the thesis. Context injection (2.5), test reliability (2.7), coordination (2.4), and the graduated autonomy model for review governance (2.3) are the primary constraints on agent effectiveness. Runtime isolation (2.1) and identity/delegation (2.2) are enabling infrastructure that must be in place for the rest to function. Legacy knowledge capture (2.6) and blast radius awareness (2.8) are cross-cutting concerns that amplify every other problem in monorepo environments specifically.

2.1 Agent Runtime: Isolation & Sandboxing

Agents need a place to run that has the tools they need and enough isolation to limit blast radius. The key architectural insight is that each agent should get an **ephemeral, isolated compute environment** with a defined toolset, filesystem, and network policy. That environment — whether a container, a VM, a Nix devshell, or something more exotic — is the trust boundary. What the agent can reach — which APIs, which repos, what network egress — is defined at the infrastructure level rather than constrained through prompting alone. Infrastructure-level controls are much harder to circumvent than prompt-level ones.

The practical implementation of this principle is a declarative workspace definition — infrastructure-as-code that specifies the agent's runtime environment, available tools, and security constraints. Specifications like devcontainers, and cloud services like GitHub Codespaces, Gitpod, and E2B, already provide variants of this pattern. Stripe's Minions (Appendix A) use pre-warmed "devboxes" — isolated environments identical to what human engineers use, but cut off from production and the internet. On a single developer machine, one to three concurrent agents is realistic; beyond that, the same workspace definition scales out to ephemeral cloud

containers. The abstraction is the same at every scale: a declared environment, provisioned on demand, torn down on completion.

The key design tensions here are isolation granularity (one container per agent vs. shared containers — more isolation means more overhead), lifecycle management (ephemeral environments are safer but slower to start; warm pools are faster but consume standing resources), and resource budgets (wall-clock time, compute cost, and API spend limits that act as circuit breakers against runaway agent loops).

2.2 Agent Identity & Delegation

The entire authentication stack — OAuth, SSH keys, API tokens — assumes a human principal or a single service account. Agents need delegated identity: an entity acting on behalf of a specific user, with specific permissions, for a bounded task, revocable at any time.

A concrete example illustrates the problem. When multiple agents commit code using a developer's personal credentials, every change appears to come from that developer. A human operating in the macro-loop needs to be able to review changes coming from agents acting on their behalf — but this review breaks most tools, since it appears that the human developer is reviewing their own code. The agents are indistinguishable from their human principal, and the workflow of independent agent work followed by human review collapses. Platform-specific workarounds exist (GitHub Apps, for instance, give agents a distinct identity with short-lived tokens), but to our knowledge there is no cross-platform standard for agent identity, nor one visibly converging. Agent identity and delegation remains one of the more consequential unsolved infrastructure problems in the space.

The design tensions include identity granularity (one identity per agent, per role, or per task — finer granularity improves auditability but increases management overhead), permission scoping (platform APIs typically offer repo-level scopes; finer constraints must live in the orchestrator), and token lifetime (short-lived tokens reduce exposure from compromise but require more frequent rotation). Beyond the workflow concern, agent identity is also a prerequisite for the measurement framework described in Section 3: if commits, CI runs, and regressions cannot be attributed to a specific agent, it becomes impossible to compute Rework Rate by source, measure Autonomy Duration, or run the controlled experiments proposed in Section 4. Identity is where the data originates; without it, the entire observability layer is blind to who — or what — produced the work. This extends beyond immediate measurement into long-term code prove-

nance: years from now, organizations will need to know which lines were written by humans versus which agents, under what context, and with what confidence level. Current monorepo "blame" tooling is not built for this, and the metadata gap will compound over time if not addressed early.

2.3 Trust Boundaries & Graduated Autonomy

Not all agent actions carry the same risk. Running tests in a sandbox is low-stakes. Pushing a commit is medium. Deploying to production is high. The system needs a way to define which actions an agent can take autonomously versus which require human approval, and to escalate dynamically when the agent encounters trouble.

This is the human-in-the-loop dial. It is not binary; it is a spectrum, and it should be configurable per action type, per agent, and per context. The goal is to minimize human interruptions for low-risk actions while maintaining human oversight for high-risk ones. Stripe's "blueprints" (Appendix A) are the most concrete public implementation of this principle: hybrid workflows that mix deterministic steps (lint, push, PR formatting — these cannot be skipped or hallucinated) with agentic steps (code generation, test interpretation — these get the full agent loop). The risk classification is embedded in the workflow definition itself, not left to runtime judgment.

The key design tensions are the action risk taxonomy (which must be environment-specific — modifying a shared utility in a monorepo is higher risk than modifying a leaf service), escalation triggers (agent has retried N times, change touches more than K files, blast radius exceeds threshold), and approval latency budget (how long an agent can wait for human approval before timing out — long waits waste agent compute, short waits pressure humans).

There is a deeper reason why graduated autonomy matters beyond efficiency: review quality degrades under volume pressure. If every agent-generated change requires human review, and an engineer is reviewing fifty agent PRs a week, the review process risks shifting from genuine scrutiny to rubber-stamping. At that point, human review is no longer a meaningful quality gate — it is a ritual that provides the appearance of oversight without the substance. This creates an ownership erosion problem: if no human wrote the code and the reviewer only skimmed it, who actually understands the system well enough to debug it when it breaks at 3 AM? Graduated autonomy is the infrastructure response — not because low-risk changes don't matter, but because routing them through human review dilutes the attention available for the changes that genuinely need it.

2.4 Coordination & Shared State

Multiple agents working on the same codebase will create merge conflicts, duplicate work, or incompatible changes. This is a distributed systems problem applied to code changes: task decomposition, dependency management, locking, and conflict resolution. What makes it worth close attention is that coordination failure has consequences that are easy to misattribute.

Consider two agents that independently modify the same file or adjacent files. Both pass tests in isolation. When merged, they conflict or interact in ways neither agent anticipated. The result is a regression that shows up in Rework Rate — but the root cause is not bad code from either agent. It is the absence of coordination between them. If an organization is measuring agent quality at the individual task level, this failure is invisible or, worse, is attributed to the agents themselves rather than to the orchestration layer that assigned overlapping work. Coordination failures are a hidden source of rework that inflates the very metric used to evaluate whether agents are producing good output.

This dynamic is also a primary mechanism behind marginal value decay (Section 3.6). With one agent, there is no coordination problem. With five, conflicts are occasional. With fifty concurrent agents on a shared codebase, the organization needs locking, dependency-aware task assignment, and a merge strategy — and the overhead of that coordination eats into the throughput gains. The inflection point on the marginal value curve — the moment when adding another agent stops helping — is largely determined by how well the coordination problem is solved.

In a monorepo, the problem is amplified. Two agents working on seemingly unrelated features can conflict if they both touch a shared dependency or a common configuration file. The build dependency graph is the only reliable way to know in advance whether two tasks are truly independent. Without dependency-aware task assignment, the orchestrator is guessing — and at agent scale, those guesses produce conflicts at a rate humans never generated, because human developers naturally coordinated through informal communication channels that agents do not have.

It is worth noting that Stripe's Minions (Appendix A) appear to sidestep this problem by design: each minion is a one-shot agent working on a pre-scoped task, and the parallel execution model works because the tasks are decomposed into non-overlapping units before agents are invoked. This is an effective approach, though it may depend on characteristics of Stripe's workflow and codebase that do not generalize to all environments — particularly

those with high implicit coupling between components, where pre-decomposing work into guaranteed-independent units is itself a hard problem.

The design tensions include task decomposition granularity (finer decomposition reduces conflicts but increases coordination overhead and the risk of cross-task dependencies), lock granularity (file-level, directory-level, or module-level — coarse locks prevent conflicts but reduce parallelism), and conflict resolution strategy (automated merge, agent-driven resolution, or human escalation — each with different latency and quality characteristics).

2.5 Code Context Injection

For smaller, self-contained repositories, context injection is rapidly becoming a solved problem through brute force. Context windows have grown large enough — and continue to grow — that an entire polyrepo-style service can often fit within a single prompt, giving the agent full visibility without any retrieval strategy at all. This is analogous to what has happened with RAG in other domains: as context windows expand, it is increasingly advantageous to simply upload entire document sets rather than engineer complex retrieval pipelines. For many teams working in polyrepo environments — particularly those with limited shared code — this shift means context injection may not require significant infrastructure investment.

Large and mature codebases are a different problem. A monorepo containing hundreds of millions or billions of lines will never fit in a prompt, and even polyrepos with extensive shared libraries face a similar challenge: the relevant context for a given task extends well beyond the repository the agent is working in. Larger context windows do not solve this. The bottleneck is not whether the model can technically *see* a million tokens — it is whether the model can *prioritize* what matters within that volume. The real infrastructure challenge is selecting and assembling the right context: using dependency graphs, language server indexing, and code intelligence to build a focused, relevant view of a codebase that may span many repositories and millions of lines. This is where context injection remains a genuine engineering problem, and where the investment matters most.

Four approaches, best used as layers in an ensemble:

Static graph-based retrieval. Walk the build dependency graph outward from the files being modified — direct dependencies, reverse dependencies, interface definitions. Deterministic, fast, and leverages existing metadata. Tradeoff: captures structural coupling but not semantic coupling.

Semantic retrieval (RAG over code). Embed the codebase into a vector store and retrieve chunks semantically similar to the task. Catches conceptual connections the graph misses. Tradeoff: code embeddings remain weaker than natural language embeddings, and keeping the index fresh at monorepo scale is a significant infrastructure challenge.

Agent-driven retrieval (search tools). Give the agent access to code search and let it explore incrementally. Most flexible, least predictable — the agent may search poorly, search too much, or fail to search for something it did not know to look for.

Curated context packages. Pre-assembled reading lists for well-understood areas of the codebase: interface contracts, key invariants, known gotchas. Machine-readable onboarding documentation. Tradeoff: requires upfront effort and ongoing maintenance; does not scale beyond high-traffic areas.

The central design tensions are precision vs. recall (graph-based is high precision, low recall; RAG is moderate on both; search is high recall, variable precision), freshness (graph traversal at HEAD is always fresh; embeddings need reindexing; curated packages need human updates), context as access control (injecting code into an agent's context effectively grants read access — context selection must respect visibility boundaries), and attention economics (more context is not always better — context injection quality is a performance variable, not just a cost optimization).

The METR study (Section 3.2) provides empirical evidence that the gap between what a developer knows and what the AI can access is a primary driver of AI-assisted slowdowns. Context injection quality is not an optimization; it is a prerequisite for positive productivity impact.

2.6 Legacy Code & Institutional Knowledge

Agents work best with code that is well-documented, follows consistent patterns, and has clear boundaries. Legacy code — especially in a large, long-lived codebase — has implicit knowledge baked into it. A function might have a subtle concurrency bug that is worked around three layers up. An API might look deprecated but have hundreds of services depending on its side effects.

Human developers absorb this institutional knowledge over months. Agents do not have access to it, and it is rarely written down in a form they can consume. This means agents operating on older parts of the codebase will produce changes that look correct in isolation but are subtly wrong in context. The review burden does not decrease —

it potentially increases, because reviewers must catch errors that a seasoned human developer would not have made.

This is not theoretical. The METR randomized controlled trial (Section 3.2) found that developers with the *deepest* familiarity with their repositories experienced the *largest* slowdowns when using AI tools. The more an expert knows that the AI does not, the more time they spend correcting its confident mistakes. A detailed case study illustrating this gap — an LLM-generated reimplementation of SQLite that compiled, passed its tests, and was 20,000x slower on a basic operation due to a single missing physical storage optimization — is included in Appendix B. The gap it reveals is not between languages or eras, but between systems built by people who measured and systems built by tools that pattern-match.

> **LLMs produce plausible architecture. They do not produce the critical details that accumulate through years of profiling against real workloads.**

In a mature monorepo maintained by thousands of engineers, these details — undocumented performance invariants, implicit conventions, and leaky abstractions — exist at orders of magnitude greater density than in any single project. This is what makes the infrastructure responses to the legacy code problem not optional but essential: knowledge capture mechanisms (architectural decision records, invariant annotations, structured code comments, curated context packages), code health scoring (automated metrics for how agent-ready a given code area is), and agent-appropriate zones (designating which areas of the codebase are safe for autonomous agent work vs. which require human-in-the-loop — data-driven, not arbitrary).

The legacy code problem is about agents misunderstanding the *past*. A related and forward-looking risk is agentic drift: what happens when agents write code that becomes the context for future agents. If agent-generated code is merged into the codebase and later used as training data, context, or few-shot examples for subsequent agent runs, there is a risk of a feedback loop in which architectural patterns become increasingly simplified — plausible but hollow, optimized for passing tests rather than for the structural invariants that make a system maintainable and performant over time. This is the generational version of the plausibility problem: one cycle of agent output may be subtly degraded; many cycles compound.

The infrastructure response to agentic drift is **architectural linting** — deterministic checks that go beyond "does it compile and pass tests" to enforce long-term structural health invariants. These might include dependency direction rules (module A may depend on module B but not the reverse), module boundary integrity checks, perfor-

mance budgets for critical paths, complexity thresholds, and naming conventions. These are invariants that experienced engineers currently enforce through review culture and institutional memory — exactly the kind of knowledge that erodes under review fatigue (Section 2.3) and that agents cannot infer from pattern-matching alone. Encoding them as automated, deterministic checks makes them durable regardless of whether the code was written by a human or an agent, and regardless of whether the reviewer scrutinized the PR or skimmed it. Architectural linting does not exist as a mature category of tooling today, but the need for it is growing in direct proportion to agent adoption. There is also a longer-term talent pipeline risk worth acknowledging: if agents handle the routine work through which junior engineers have traditionally built institutional knowledge — bug fixes, small features, dependency updates — the apprenticeship path that produces the next generation of senior engineers is disrupted. Architectural linting and knowledge capture mechanisms become important not only for agent quality, but for human learning as well.

2.7 Test Reliability & the Flaky Test Multiplier

Flaky tests sit at the intersection of several problem areas and their impact is amplified at agent scale. The core agent workflow is *iterate until tests pass.* A flaky test breaks this loop entirely — not by producing a wrong answer, but by introducing stochastic noise into the feedback signal the agent depends on to converge toward a working solution. An agent iterating toward a "passing" state cannot distinguish a flake from a real failure, so it either retries indefinitely (burning compute), makes unnecessary code changes to fix a test that is not actually broken (introducing regressions), or gives up and escalates (defeating the purpose of autonomy). In each case, the agent fails to converge — not because its code is wrong, but because the signal it is optimizing against is unreliable.

The title of this section uses the word "multiplier" deliberately. If an agent's workflow requires passing through multiple sequential test stages to produce a final PR, and each stage has an independent flake rate, the probability of completing the entire workflow without a false failure drops rapidly. At a 5% flake rate per stage across 10 stages, the probability of a clean run is $0.95^{10} \approx 60\%$. At 10% flake rate, it falls to $0.9^{10} \approx 35\%$. For a human developer, a flaky test is an annoyance — re-run it and move on. For an agent running autonomously, flake rate compounds across every stage of the workflow and becomes the dominant factor in whether the agent can complete a task without human intervention. Flaky tests go from being a developer annoyance to being an **agent-halting condition.** The

ROI on test reliability investments — which has always been hard to justify in dollar terms — suddenly becomes concrete: every percentage point of flakiness translates directly into wasted agent compute cycles and failed autonomous workflows.

The critical design questions are whether the agent's workflow has access to known-flaky-test databases (so it can distinguish flake failures from real failures), what the retry policy is (how many retries before escalation, and whether retries cost against the agent's resource budget), and whether agent-triggered tests run in fully isolated environments (to eliminate shared-state flakiness sources). Stripe's approach (Appendix A) offers a concrete reference and illustrates a deeper principle: their combination of local linting in under five seconds, selective test execution from a battery of over three million tests, and autofixes for known failure patterns functions as a prefilter that reduces the state space the agent must navigate. Rather than presenting the full test suite as a black box that returns pass or fail, Stripe's infrastructure transforms it into a guided environment — catching trivial failures before CI, narrowing the test surface to what is relevant, and resolving known issues deterministically. The agent's iteration loop operates on a cleaner signal, which is why a hard cap of two CI rounds is sufficient rather than punitive.

2.8 Blast Radius Awareness

Agents operating in a monorepo need blast radius awareness — automated impact analysis — at the infrastructure level, not the prompt level. The build dependency graph already knows the fan-out of any given file: how many targets it affects, how many tests it triggers, how many teams own downstream dependencies. This information can be surfaced to the agent *before* it commits to an approach, not after it triggers a million test targets.

The idea is to **turn the monorepo's own dependency metadata into a cost signal that shapes agent behavior**, the same way build time and review latency shape human behavior today. A human developer instinctively avoids modifying widely-depended-on code unless necessary. Agents need the equivalent instinct, delivered as data.

A concrete design sketch: before the agent finalizes a change, the system queries the dependency graph for the fan-out and returns a score and summary ("this change affects 347 downstream targets across 12 teams"). Low blast radius changes proceed autonomously. Medium blast radius triggers additional test coverage requirements. High blast radius triggers human review before any tests run. If the blast radius is high, the system could suggest alternative approaches

that achieve the same goal with narrower impact — a local wrapper instead of modifying a shared utility, for example.

Critically, this moves risk assessment into the agent's *planning* phase rather than its *execution* phase. Without blast radius awareness, the agent discovers that its approach has wide impact only after it has written the code, triggered a long build, and watched CI fail or flag across dozens of downstream targets — potentially hours of wasted compute. With blast radius awareness, the agent pre-calculates impact before writing a line of code, and can redirect toward a narrower approach when the cost is near zero. This is a significant factor in agentic ROI: the most expensive agent failure is one that consumes a full work cycle before being caught.

Summary: The Problem Landscape at a Glance

Problem Area	Why It Matters at Agent Scale	Monorepo Amplifier	Primary Failure Mode	Likely First Experiment
2.1 Runtime & Isolation	Agents need trust boundaries that can't be prompt-engineered away	Shared infrastructure under higher load	Runaway loops, cross-contamination	Measure container startup vs. task time; baseline resource budgets
2.2 Identity & Delegation	Agents are invisible to review/audit tools without distinct identity	Thousands of agents on one codebase, no attribution	Broken review workflows, unmeasurable outcomes	Deploy app-based agent identities; measure review friction
2.3 Graduated Autonomy	Human review cannot scale to agent volume; review quality erodes	Higher-risk changes require more nuanced classification	Rubber-stamping, ownership erosion	Classify actions by risk tier; measure escalation rates

Problem Area	Why It Matters at Agent Scale	Monorepo Amplifier	Primary Failure Mode	Likely First Experiment
2.4 Coordination	Concurrent agents produce merge conflicts and hidden rework	Implicit coupling means "unrelated" tasks can conflict	Rework misat-tributed to agent quality rather than orchestration	Dependency-aware task assign-ment; measure conflict rate vs. agent count
2.5 Context Injection	Agents produce locally-correct but globally-wrong changes without adequate context	Monorepos cannot be brute-forced into a context window	Plausible but subtly broken code	Compare retrieval strategies (graph, RAG, search, ensemble)
2.6 Legacy Code & Inst. Knowledge	Institutional knowledge lives in developer culture, not in docu-mentation agents can access	Decades of undocu-mented invariants at massive density	Expert time consumed correcting confident mistakes	Code health scoring; curated context packages for high-traffic areas
2.7 Test Reliability	Flaky tests prevent agents from con-verging; flake rate compounds across stages	Larger test suites = more flake exposure	Agent-halting condition; wasted compute on false failures	Flake detection integra-tion; measure iterations-to-green

Problem Area	Why It Matters at Agent Scale	Monorepo Amplifier	Primary Failure Mode	Likely First Experiment
2.8 Blast Radius	Agents lack intuition about down-stream impact of changes	Single change can cascade across thousands of targets	Global rebuilds triggered by unin-formed agents	Surface de-pendency fan-out data; compare incident rates with/without

3. Measuring Agent Utility

Any team operating in this space must be rigorous about measurement, and measurement in agentic systems is harder than it appears. A foundational principle from the developer productivity research community (Forsgren, et al.) applies with even more force in the agentic era: **all metrics are proxies.** A metric only matters because of what it represents — a signal about system health, team effectiveness, or business value. Engineering is the practical application of science, and measurement is what separates it from mere craft. This principle is the guard rail against the considerable temptation to measure what agents make easy to count rather than what actually matters.

3.1 The Activity Trap

The most important measurement anti-pattern to name up front: organizations are already regressing to Taylorism by measuring AI prompt counts, lines of code generated, and PR volume. This is the **Activity Trap** — measuring low-value, easily-gamed outputs instead of meaningful outcomes.

Agents make this trap worse because they can generate enormous volumes of activity. An agent that opens twenty PRs in a day looks productive by activity metrics. If fifteen of those PRs require multiple review rounds, three introduce regressions, and two are reverted — the net outcome may be negative. Activity metrics for agents are not just uninformative; they are actively misleading because they create incentives to optimize for volume over value.

A vivid illustration emerged in March 2026 under the label **"token-maxxing"** — a status competition among engineers to consume the most AI tokens. At some companies, employees compete on internal leaderboards tracking token consumption; at others, managers factor raw AI usage into performance reviews. Individual engineers report consuming billions of tokens per week by running swarms of parallel agents around the clock. The tokenmaxxing phenomenon is the Activity Trap in its purest and most expensive form. Token consumption measures *input* — resources burned — not *output*, let alone *outcomes*. The leaderboards do not track whether the code produced is correct, maintainable, or even used. The article that popularized the term asked the question this paper's measurement framework exists to answer: "Are any of these tokenmaxxers producing anything good?" Without Rework Rate, regression tracking, quality-adjusted throughput, and the other outcome-level metrics described in this section, that question is unanswerable — which is precisely the problem.

The corrective is to measure at the system and outcome level, not the activity level. The established frameworks — DORA, SPACE, and DevEx — provide the scaffolding for this, though each requires adaptation for the agentic context.

3.2 The Perception Gap: Evidence from the METR Study

A 2025 randomized controlled trial by METR (Model Evaluation & Threat Research) provides the most rigorous empirical caution available about AI-assisted development productivity. The study deserves close attention — not because it settles the question, but because it exposes exactly the kind of measurement failure this paper warns against.

The study. METR recruited 16 experienced open-source developers and had them complete 246 real tasks (bug fixes, features, refactors) on their own repositories — mature codebases averaging over 1 million lines of code that the developers had worked on for an average of 5 years. Tasks were randomly assigned to allow or disallow AI tools. When AI was allowed, developers primarily used Cursor Pro with Claude 3.5/3.7 Sonnet. All sessions were screen-recorded and self-reported times were validated.

The result. Developers using AI tools were 19% *slower* at completing tasks. Not faster. Slower.

The perception gap. Before the study, developers forecasted that AI would make them 24% faster. After completing the study — after experiencing the actual slowdown — they still estimated they had been 20% faster. That is a 40+ percentage point gap between

perceived and actual productivity. Developers were not just wrong about the magnitude of AI's benefit; they were wrong about its *direction*.

Why it happened. METR's factor analysis identified several contributing causes that map directly to this paper's problem landscape: a context familiarity penalty (developers who knew their codebases most deeply were slowed down most — the institutional knowledge problem from Section 2.6), high quality standards as friction (code that is "functionally correct" but fails implicit quality standards requires substantial human cleanup — the Rework Rate problem), and attention fragmentation (AI tools introduced micro-interruption patterns that disrupted flow state — the DevEx Cognitive Load problem).

What this validates. The Activity Trap is empirically real: activity metrics and outcome metrics can point in opposite directions. Self-reported productivity is unreliable: the 40-point perception gap is the strongest argument for instrumented measurement over developer surveys. Context injection quality is a performance variable: the gap between what the developer knows and what the AI can access is a primary driver of slowdowns. The downstream pipeline matters more than code generation speed: the AI generated code fragments quickly, but the total task time was longer — the Narrow Pipe thesis observed in a controlled experiment.

What complicates the picture. The study measured Level 2 (attended pair programming), not Level 3-5 (unattended agentic). The slowdown may be specific to the attended workflow where context-switching and correction overhead dominate. Unattended agents that run asynchronously and deliver completed PRs may avoid the attention-fragmentation problem — but this is a hypothesis, not a demonstrated fact. More broadly, the METR result is strongest as evidence for "don't trust subjective productivity impressions" and "context gaps matter in mature codebases." It is weaker as direct evidence for this paper's claim that higher-autonomy pipelines are where value will emerge — that claim rests on architectural reasoning and the Stripe case study, not on the METR data. Also: experienced developers on familiar codebases is the hardest case, and it is exactly the case that mature monorepo environments present. This should temper optimism about agent-driven productivity gains in precisely the environments where this paper proposes deploying agents.

The implication. The METR study does not argue against agentic development. It argues for *rigorous, instrumented, outcome-based measurement* of agentic development. The 40-point perception gap is one of the strongest empirical arguments for why the measurement framework in this section exists.

Without it, organizations will deploy agents, feel faster, and never discover they are slower — until Rework Rate and regression data make the cost undeniable.

3.3 Established Frameworks, Adapted for Agents

DORA: The Five Keys of Delivery (2026 Edition) The DORA metrics remain the gold standard for measuring delivery performance at the system level. The 2026 addition of **Rework Rate** as the fifth metric is particularly significant for agentic workflows.

Metric	Definition	Agentic Implication
Deployment Frequency	How often code is successfully pushed to production.	Agents can inflate this metric trivially. Must be paired with quality gates.
Lead Time for Changes	Time from first commit/intent to production.	Stable agent micro-loops (fewer iterations, better context) directly reduce lead time. The metric most improved by good agent infrastructure.
Change Failure Rate	% of deployments requiring rollback or hotfix.	The critical quality gate. If agents produce code that fails this gate, the velocity gain is lost.
Failed Deployment Recovery Time	How quickly the system is restored after a failure.	Agents could potentially accelerate recovery, but this is speculative.

Metric	Definition	Agentic Implication
Deployment Rework Rate (Added 2024, benchmarked 2025)	**The ratio of deployments that are unplanned but happen as a result of an incident in production.**	**The most revealing DORA metric for evaluating agent quality.** If agent-generated code drives up rework rate, the speed gain is illusory. Tracking rework rate by source (agent vs. human, by agent type, by code area) provides the most direct signal of where agent quality is sufficient and where it is not.

SPACE: The Five Dimensions of the Developer SPACE provides a holistic view that resists single-metric optimization. In the agentic context:

Dimension	Agentic Adaptation
Satisfaction & Well-being	Are developers experiencing agent-related frustration (bad reviews, regressions to fix, trust erosion)? Or are agents genuinely reducing toil? Survey-based; cannot be inferred from system data.
Performance	Outcome-based quality and impact: task completion rate, regression rate, quality-adjusted throughput.
Activity	Commits, reviews, PRs. **Use only as context, never as a target.** This is where the Activity Trap lives.

Dimension	Agentic Adaptation
Communication & Collaboration	How effectively humans and agents share information: quality of agent-generated PR descriptions, signal-to-noise ratio in agent escalations.
Efficiency & Flow	If agents generate more interrupts (review requests, escalations, incident responses) than they absorb, they are net-negative on this dimension.

DevEx: The Three Pillars of Experience

Pillar	Agentic Analog
Feedback Loops	CI/CD latency, test execution time, code search speed. These are the micro-loop bottlenecks. Faster feedback loops mean fewer wasted agent iterations.
Cognitive Load	The mental effort required to orchestrate, review, and correct agents. If developers spend more cognitive effort managing agents than doing the work themselves, the tool is net-negative.
Flow State	The promise of agents is to increase flow by absorbing toil; the risk is that they decrease flow by generating review burden and escalations.

3.4 Operational Observability

Before optimizing the pipeline for agent-scale throughput, we need to see the problem concretely. This is the instrument-before-you-optimize principle. Operational observability answers the question: what are agents doing, and what is it costing?

This means tracking resource attribution (which agents trigger which builds, how much CI resource is consumed per agent task, where the retry loops are happening), pipeline throughput (queue depth and latency at each stage for agent-generated

vs. human-generated changes), and failure analysis (what fraction of agent-initiated work fails, at what stage, and why — categorized by root cause: flaky tests, context gaps, permission errors, resource exhaustion, coordination conflicts).

This measurement layer is low-risk, immediately useful, and provides the data to prioritize everything else. It should be the first investment.

3.5 Agent-Specific Effectiveness Metrics

Beyond the established frameworks, agentic development introduces metrics that have no human-era equivalent:

Metric	Description
Autonomy Duration	How far an agent progresses through meaningful work before requiring human intervention. The primary measure of macro-loop efficiency.
Task completion rate	Fraction of assigned tasks completed without human intervention. Track by task type, code area, and complexity tier.
Time-to-merge	Elapsed time from task assignment to merged PR, including review cycles. Compare to human baseline.
Review revision rate	Review round-trips per agent PR. High rate suggests poor context or quality; near-zero may suggest insufficient review rigor — or rubber-stamping under volume pressure (see Section 2.3).
Micro-loop efficiency	Average iterations per task, by task type. A proxy for how well the agent environment supports autonomous work.

Metric	Description
Blast radius incidents	Changes that triggered unexpected downstream failures. Even if caught in review, these indicate context injection failures.

Metrics That Mislead: Lines of code produced is the most obvious vanity metric. "Compiles and passes tests" is equally dangerous: the LLM-generated SQLite rewrite described in Appendix B compiled, passed its tests, and was 20,000x slower than the original on a basic operation. Only performance benchmarks — outcome-level measurement — revealed the gap. PRs per day and raw cost-per-PR are similarly misleading without downstream quality accounting. Per the SPACE framework, activity metrics should be used only as diagnostic context, never as targets.

What remains unmeasured: Ownership attribution — who understands a given piece of code well enough to debug and maintain it — has no standard metric in any current framework. In a world where agents produce code and humans review it under volume pressure, the assumption that "the reviewer owns it" becomes increasingly fragile. This is an open problem that the field has not yet solved.

3.6 Value & ROI Measurement

The hardest and most important layer. Value measurement answers: are agents making the engineering organization more effective? A simple framing helps structure the analysis:

Net agent value ≈ throughput gain − review burden − CI/compute cost − regression/rework cost − coordination overhead

This is not a formula to compute precisely. It is a thinking tool that makes the cost structure visible and prevents the common mistake of measuring only the numerator (throughput) while ignoring the denominator (everything else). Each component maps to a concrete measurement:

Developer time recaptured. If agents handle routine tasks, how much developer time is freed for higher-leverage work? This must be measured by what developers actually do with the freed time, not just the time saved.

Cycle time compression. For a given type of change, what is the end-to-end cycle time with agent assistance vs. without? This cap-

tures the full pipeline effect, not just code generation speed. Maps directly to DORA's Lead Time for Changes.

Infrastructure cost of agents. The total cost of running agents: compute, API tokens, CI resources consumed, and the human time spent reviewing agent output, fixing agent mistakes, and maintaining agent infrastructure. This is the denominator in any ROI calculation.

Quality-adjusted throughput. A composite metric that weights throughput by quality (DORA's Change Failure Rate and Rework Rate, code health scores, review feedback). An agent that produces five low-quality PRs requiring multiple review rounds is less valuable than one that produces two clean PRs that merge on first review.

Marginal value decay. As agent count increases, does each additional agent produce proportional value? Or do coordination costs, CI contention, and review bottlenecks cause diminishing returns? Plotting value per agent against agent count reveals the inflection point where adding agents stops helping.

---

4. From Observation to Experimentation

This paper has mapped a problem landscape (Section 2) and a measurement framework (Section 3). The question that remains is: what do we do about it?

The answer is not to guess which solutions work and build them. It is to treat each problem area as a site for **structured experimentation** — formulating hypotheses, designing controlled tests, measuring outcomes, and iterating. The developer productivity field has decades of experience with this methodology; the agentic era introduces new variables but not a new epistemology. We still instrument before we optimize. We still measure outcomes, not activity. We still define counterfactuals and control for confounds.

4.1 Why Experimentation, Not Roadmaps

The agentic development space is moving too fast for traditional multi-quarter roadmaps. Models improve on a monthly cycle. Tool capabilities shift with each release. Best practices that hold in February may be obsolete by June. In this environment, the teams that make progress are the ones that can run experiments quickly, measure results rigorously, and pivot based on evidence.

This is a natural fit for organizations with an experimentation culture.

The ability to frame agent infrastructure as a series of measurable experiments — rather than speculative bets — is a structural advantage. Each experiment produces knowledge regardless of whether the hypothesis holds, because a negative result narrows the search space.

4.2 The Experimental Agenda

Each problem area from Section 2 maps to at least one high-signal experiment. The following are prioritized by a combination of expected impact, measurement feasibility, and control over variables.

Experiment 1: Observability and cost attribution (first priority). Instrument the development pipeline to attribute builds, test runs, and CI resource consumption to specific agents and tasks. This follows the instrument-before-you-optimize principle and is the prerequisite for every subsequent experiment. The measurement is straightforward, the risk is low, and the data is immediately useful for prioritization.

Experiment 2: Flake detection integration. Compare agent test-loop behavior with and without access to known-flaky-test databases. Measure iterations-to-green, false regression rate, total compute spend, and escalation frequency. This is a high-signal experiment because the variables are controllable and the outcomes are directly measurable. It is likely the highest-leverage single improvement to agent workflow reliability.

Experiment 3: Context injection strategies. Compare agent task success rates across different context injection approaches (graph-only, RAG-only, search-only, ensemble). Measure token spend vs. task quality to find the efficient frontier. The METR study provides the baseline hypothesis: context injection quality is a primary driver of agent effectiveness in mature codebases.

Experiment 4: Blast radius estimation. Surface dependency fan-out data to agents before they commit to an approach. Compare blast radius incident rates with and without this signal. This is the monorepo-specific experiment that turns an existing infrastructure asset (the dependency graph) into an agent safety mechanism. It is also a compute economics experiment: in a monorepo, preventing a single agent from inadvertently triggering a global rebuild is a direct and measurable cost saving.

Experiment 5: Agent identity and review workflow. Deploy app-based agent identities (e.g., GitHub Apps) and measure impact on review cycle time, approval friction, and audit clarity. Track identity-related incidents (permission errors, token expiry during tasks) to

quantify the operational cost of current workarounds.

4.3 Experimental Design Principles

Measuring agent value rigorously requires experimental discipline:

Use A/B or holdout designs where possible. Teams with agent access vs. without, on comparable workstreams. This controls for the Hawthorne effect and avoids attributing general productivity trends to agents.

Measure leading and lagging indicators together. Leading: task completion rate, time-to-merge. Lagging: regression rate, code health trends, developer satisfaction. Divergence between leading and lagging indicators (fast merges but rising regressions) is a critical warning signal.

Define the counterfactual explicitly. Is the comparison "agent vs. no agent" or "agent vs. human doing the same task"? The former measures absolute value; the latter measures comparative advantage. Both matter but they answer different questions.

Account for learning effects. Agent effectiveness improves as teams refine task definitions, context packages, and prompt strategies. Early measurements will understate long-run value. Build time-series analysis into the measurement plan.

4.4 The Platform Opportunity

The experimental agenda points toward a central conclusion:

> **The future tooling challenge is not building agent features on top of the existing developer platform. It is rebuilding the developer platform for a world where agents are first-class participants.**

This includes build and test systems that understand agent-triggered runs differently from human-triggered runs (priority, resource quotas, cost attribution); a monorepo permission and blast-radius model that constrains agent scope at the infrastructure level; flaky test detection integrated into the agent loop; and code context systems that make institutional knowledge machine-readable.

Most current tooling (Claude Code, Copilot Workspace, Cursor) optimizes the *inner loop* — an agent working autonomously on a bounded task. The orchestration, identity, review, and deployment layers are the *outer loop.* The outer loop — where multi-agent coordination, trust, and governance problems live — is where the real platform op-

portunity lies. The experiments described above are the first steps toward building it with evidence rather than intuition.

A common thread runs through several of the problem areas surveyed in Section 2: graduated autonomy with deterministic workflow nodes (2.3), architectural linting to prevent agentic drift (2.6), flake detection to ensure reliable feedback signals (2.7), and blast radius estimation to catch high-impact changes during planning (2.8) are all instances of the same pattern. Each replaces a human judgment gate — which cannot scale to match agent throughput — with a deterministic infrastructure gate that operates automatically and consistently regardless of volume. As agent autonomy increases, this verification layer must grow in proportion. The organizations that operate safely at Levels 3 and 4 will be those that have invested in making verification as automated and rigorous as code generation has become — ensuring that agents do not merely pass tests, but adhere to the structural, performance, and security invariants that define a healthy system over time.

4.5 What Leaders Should Do Now

For readers looking to convert this paper into action, the following is a prioritized synthesis:

Act now: Instrument the development pipeline for agent attribution and cost visibility. This is zero-risk, immediately useful, and the prerequisite for every decision that follows. If commits, CI runs, and regressions cannot be traced to specific agents, no subsequent measurement or experiment is possible.

Next quarter: Run the flake detection and context injection experiments (Section 4.2, Experiments 2 and 3). These are controlled, measurable, and address the two highest-leverage constraints on agent effectiveness in mature codebases.

Avoid: Leaderboards, token consumption metrics, or any measurement system that rewards activity over outcomes. These create perverse incentives and generate misleading signal. The Activity Trap (Section 3.1) is not a theoretical risk — it is already happening in the industry.

Readiness signals for Level 3/4 operation: CI feedback loops under 10 minutes. Flake rate under 5%. Dependency-aware task decomposition in the orchestrator. Agent identity infrastructure that supports attribution and review workflows. Without these, higher-autonomy workflows will generate more cost than value.

The long game: The organizations that benefit most from agentic development will not be those that deploy the best model first. They

will be those that redesign verification, attribution, coordination, and trust boundaries for a world where code generation is cheap and downstream judgment is scarce.

References

- Forsgren, N., Humble, J., & Kim, G. *Accelerate: The Science of Lean Software and DevOps.* IT Revolution Press, 2018. (DORA foundation)
- Forsgren, N., Storey, M-A., et al. "The SPACE of Developer Productivity." *ACM Queue*, 2021. (SPACE framework) https://queue.acm.org/detail.cfm?id=3454124
- Farley, D. & Forsgren, N. "The BEST Way To Measure Software Developer Performance." (Discussion: all metrics are proxies; measurement separates engineering from craft) https://www.youtube.com/watch?v=k8-DzU5py_I
- Smith, S. "The Truth About Developer Productivity in the AI Age." (Discussion: the Activity Trap and why outcome-level measurement is essential) https://www.youtube.com/watch?v=kDBeFOscZpc
- Shapiro, D. "The Five Levels: From Spicy Autocomplete to the Dark Factory." January 2026. (The autonomy spectrum taxonomy; NHTSA-modeled five levels of AI coding automation) https://www.danshapiro.com/blog/2026/01/the-five-levels-from-spicy-autocomplete-to-the-software-factory/
- StrongDM / McCarthy, J., Taylor, J., & Chauhan, N. "Software Factories and the Agentic Moment." 2025-2026. (Dark Factory reference implementation; spec-driven development with holdout scenario testing) https://simonwillison.net/2026/Feb/7/software-factory/
- Stanford CodeX. "Built by Agents, Tested by Agents, Trusted by Whom?" February 2026. (Legal and governance analysis of Dark Factory implications) https://law.stanford.edu/2026/02/08/built-by-agents-tested-by-agents-trusted-by-whom/
- Becker, J., Rush, N., Barnes, E., & Rein, D. "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity." METR, July 2025. arXiv:2507.09089. (RCT finding 19% slowdown; 40-point perception gap between believed and actual AI productivity impact) https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/
- Roose, K. "Tokenmaxxing." *The New York Times*, March 20, 2026. (The Activity Trap as cultural phenomenon: token consumption leaderboards, AI usage in performance reviews, and the unanswered question of output quality) https://www.nytimes.com/2026/03/20/technology/tokenmaxxing-

ai-agents.html
- "Your LLM Doesn't Write Correct Code. It Writes Plausible Code." *Vagabond Research*, March 2026. (LLM-generated SQLite rewrite: 20,171x slower on primary key lookup; the gap between plausible architecture and performance-critical invariants) https://blog.katanaquant.com/p/your-llm-doesnt-write-correct-code
- Stripe Engineering. "Minions: Stripe's One-Shot, End-to-End Coding Agents." Parts 1 & 2, February 2026. (Enterprise-scale agentic development case study; 1,000+ PRs/week) https://stripe.dev/blog/minions-stripes-one-shot-end-to-end-coding-agents

Appendix A: Case Study — Stripe Minions

Stripe's "Minions" system (publicly documented February 2026) is the most detailed case study of enterprise-scale agentic development available. The system produces over 1,000 merged pull requests per week — with humans reviewing the code but writing none of it. This appendix uses Minions as a **reference implementation of certain outer-loop design principles** — not as proof that those principles generalize unchanged to mature monorepos. Stripe's codebase is large but operates under different constraints than a monorepo at the scale discussed in the body of this paper. Where the architecture illustrates a principle from Section 2, we note it; where it leaves monorepo-specific problems unaddressed, we note that too.

Sources: Stripe Engineering Blog (Parts 1 & 2), ByteByteGo analysis, Ry Walker Research.

Architecture Overview

Engineers invoke agents via Slack, CLI, or web. Agents run in isolated pre-warmed devboxes, access 400+ internal tools via MCP, and produce PR-ready code within 2 CI rounds. The system is built on a fork of Block's open-source Goose agent, extended with deep integrations into Stripe's internal infrastructure.

The flow: invocation → isolated devbox (10-second spin-up) → MCP context hydration → agent loop → local lint → CI (max 2 rounds with auto-fix) → pull request for human review.

Mapping to the Problem Landscape

Runtime & Isolation (Section 2.1). Devboxes are identical to what human engineers use, but isolated from production and the

internet. Each agent gets a full, isolated environment — high overhead per agent, mitigated by warm pools with 10-second spin-up. Resource budgets are enforced through the CI round cap: at most two CI runs, then human escalation. This is a concrete implementation of the circuit-breaker pattern.

Context Injection (Section 2.5). Stripe combines pre-hydration (relevant MCP tools run over likely-looking links before the agent loop begins), tool-based retrieval (Sourcegraph code search, documentation, ticket systems via the "Toolshed" MCP server), and scoped agent rules (conditional rules based on subdirectories rather than blanket global instructions). This is the layered ensemble approach the paper recommends.

Graduated Autonomy (Section 2.3). Stripe's "blueprints" mix deterministic and agentic steps. The "implement the feature" step gets the full agentic loop. The "run linters" step is hardcoded. The "push the branch" step is hardcoded. The risk classification is embedded in the workflow definition — infrastructure-level governance of agent behavior.

Test Reliability (Section 2.7). Local linting in under five seconds, selective test execution from over three million tests, autofixes for known failure patterns, and a hard cap of two CI rounds. The autofix for known failure patterns is institutional knowledge encoded as deterministic remediation.

Identity (Section 2.2). The public documentation does not detail how Minions handle identity and code attribution. This is a notable gap at Stripe's scale (1,300+ PRs per week).

What Stripe Validates

Infrastructure is the moat, not the model. The primary reason the Minions work has almost nothing to do with the AI model powering them. It has everything to do with the infrastructure that Stripe built for human engineers, years before LLMs existed. Decades of developer productivity investment *are* the foundation for agent-scale development. Organizations without that foundation cannot replicate this by deploying a better model.

Context engineering > prompt engineering. The quality of context injection is a primary lever for agent effectiveness — not the sophistication of the prompt or the agent loop itself.

Deterministic guardrails > behavioral guardrails. Blueprints enforce correct behavior through workflow structure, not through instructions to the agent.

Hard limits prevent runaway costs. The max-2-CI-rounds cap directly prevents the unbounded retry loop this paper identifies as a critical risk.

Monorepo-Specific Gaps

Stripe's codebase is large but it is not a mature monorepo at the scale where a single utility change cascades across thousands of independent products. Blast radius awareness (Section 2.8), cross-agent coordination at monorepo scale (Section 2.4), and CI infrastructure under orders-of-magnitude-greater agent load remain open problems.

Stripe's Observed Settings

Variable (from Section 2)	Stripe's Setting	Notes
Isolation granularity	One full devbox per agent	High isolation, mitigated by warm pools
Resource budgets	Max 2 CI rounds, then stop	Hard cap; human escalation on failure
Tooling surface	~500 MCP tools via Toolshed	Broad but curated; default subset per agent
Context strategy	Pre-hydration + search + scoped rules	Layered ensemble
Action risk taxonomy	Blueprint: deterministic vs. agentic nodes	Risk embedded in workflow structure
Escalation triggers	2 CI failures without resolution	Simple, binary
Freshness	Live MCP queries at run time	Always-fresh; no stale embeddings

---

Appendix B: Case Study — The Plausibility Gap (LLM-Generated SQLite)

This case study illustrates the gap between plausible and correct code described in Section 2.6. It is drawn from a March 2026 analysis published by Vagabond Research.

The Project

A developer used LLM agents to produce a ground-up Rust reimplementation of SQLite — 576,000 lines across 625 files, with a parser, query planner, VDBE bytecode engine, B-tree, pager, and WAL. The code compiled. It passed its tests. It read and wrote the correct SQLite file format. Its README claimed MVCC concurrent writers, file compatibility, and a drop-in C API. By every activity metric, it looked like a working database engine.

The Result

A basic primary key lookup on 100 rows took 1,815 milliseconds — compared to 0.09 milliseconds in SQLite. The rewrite was 20,171x slower on one of the most fundamental database operations.

The Root Cause

The failure was not a syntax error or a missing feature. It was a leaky abstraction. The LLM understood the B-tree data structure and generated a correct implementation of it. What it did not understand was the physical storage optimization that links a logical column declared as `INTEGER PRIMARY KEY` to the internal rowid — the actual B-tree key. In SQLite, this is the rowid alias: `WHERE id = 5` on an `INTEGER PRIMARY KEY` column does not scan rows — it goes directly to the B-tree key in O(log n).

The LLM-generated implementation's `is_rowid_ref()` function only checked for three literal strings: `rowid`, `_rowid_`, and `oid`. A column declared as `id INTEGER PRIMARY KEY`, even when flagged internally as `is_ipk: true`, never triggered the B-tree fast path. Every `WHERE id = N` query ran a full table scan. At 100 rows with 100 lookups, that is 10,000 comparisons instead of roughly 700 B-tree steps.

This check exists in SQLite because someone — probably Richard Hipp, decades ago — profiled a real workload, found that named primary key columns were not hitting the B-tree search path, and wrote one line in `where.c` to fix it. The line is not fancy. It does not appear in any API documentation. No LLM trained on documentation and Stack Overflow answers will know about it.

Why This Matters for Mature Codebases

The gap is not between C and Rust, or between old and new. It is between systems that were built by people who measured, and systems that were built by tools that pattern-match. SQLite is a single-developer project with 26 years of history. A mature monorepo main-

tained by thousands of engineers over a similar timespan contains orders of magnitude more of these details — undocumented performance invariants, implicit conventions, and leaky abstractions that carry correctness, invisible to any tool that pattern-matches rather than measures.

This case study is an illustration of a class of failures, not proof that all LLM-generated code exhibits this degree of degradation. But the pattern it reveals — plausible architecture with missing critical details — is the failure mode that the institutional knowledge problem (Section 2.6) and architectural linting proposal exist to address.

Source: "Your LLM Doesn't Write Correct Code. It Writes Plausible Code." Vagabond Research, March 2026.

Author's note: AI tools were used extensively at every stage of this paper, including synthesis, structural editing, prose revision, and critique. I treated them as accelerants and adversarial readers rather than authorities. I am the sole author of this paper and am responsible for its claims, evidence, interpretations, and conclusions.
