ONA

How Stripe and Ramp Built Self-Driving Codebases

Background agents, infrastructure primitives, and the next era of software delivery

MARCH 2026

CONTENTS

Executive Summary

Imagine a codebase that drives itself. Code is written, reviewed, tested, and deployed continuously and autonomously. Engineers are not writing every line. They are setting constraints, reviewing outcomes, and designing the systems that do the work.

This is not speculative. Stripe, Ramp, and Spotify have already built it.

Stripe's Minions platform merges over 1,300 agent-generated pull requests per week. Ramp's Inspect agent accounts for over half of all merged PRs across their frontend and backend repositories. Spotify's background coding agents have produced over 1,500 merged pull requests, cutting migration time by 60–90%.

These organisations did not get there by giving developers better coding assistants. They got there by building infrastructure that lets agents run autonomously: triggered by events, governed by policy, connected to internal systems, and coordinated across hundreds of repositories.

Most engineering organisations have not made that leap. They adopted coding agents (Copilot, Cursor, Claude Code) and saw individual developers get faster. But the 2025 DORA State of AI-Assisted Software Development found that organisational delivery metrics (lead time, deployment frequency, change failure rate) show inconsistent improvement. Elite-performing teams were the least likely group (31%) to report that AI tool adoption improved their organisational outcomes.

Goldratt's theory of constraints explains why. Coding agents solved the code-generation bottleneck. The constraint shifted downstream: code review, testing, CI maintenance, dependency management, security patching, release coordination. Accelerating code generation while leaving these bottlenecks untouched is local optimisation without system improvement.

Background agents address the entire delivery system. They run in their own cloud environments, triggered by events rather than humans, scoped by governance rather than trust, and coordinated across the organisation rather than confined to a single repo.

This paper covers what background agents are, how the leading engineering teams built their infrastructure, what primitives are required, internal data from Ona's own deployment, and why most organisations will need to buy rather than build.

TL;DR

1. **Individual speed does not equal organisational velocity.** Coding agents made developers faster. Background agents make the delivery system faster.
2. **The false summit is real.** Running multiple coding agents in parallel on your laptop feels like transformation. It is not. There is no governance, no event triggering, no fleet coordination.
3. **Five infrastructure primitives separate a demo from a deployment.** Sandboxed execution, governance, context and connectivity, triggers, and fleet coordination.
4. **The companies doing this invested years in infrastructure.** Stripe, Ramp, and Spotify built custom platforms because they had the scale and engineering talent to justify it.
5. **For everyone else, the realistic path is to buy.** Building your own background agent infrastructure is a multi-quarter, multi-team investment. Most organisations should not attempt it.

---

1. What Is a Background Agent?

A coding agent needs your machine and your attention. A background agent needs neither. It runs in its own cloud-hosted development environment with a full toolchain, test suite, and network access. It is triggered by events, not humans. It produces pull requests. A human reviews before merge.

The differences from coding agents are architectural, not cosmetic:

CODING AGENTS VS. BACKGROUND AGENTS

Property	*Coding agent*	*Background agent*
Execution environment	Developer's local machine	Cloud VM or container
Invocation	Developer initiates each run	Event-driven: schedule, webhook, PR, Slack
Repository scope	Single repo per session	Fleet: one agent per repo, in parallel
Developer's role during execution	Active, present	Not required
Session continuity	Ends with machine/session	Runs independently
Credentials	Inherits developer's credentials	Scoped identity with least-privilege permissions
Audit trail	None by default	Logged per run

2. Production Deployments

Three engineering organisations have publicly documented background agent deployments at enterprise scale. Each account below is based on published engineering blog posts and, in the case of Ramp, a detailed infrastructure case study published by Modal.

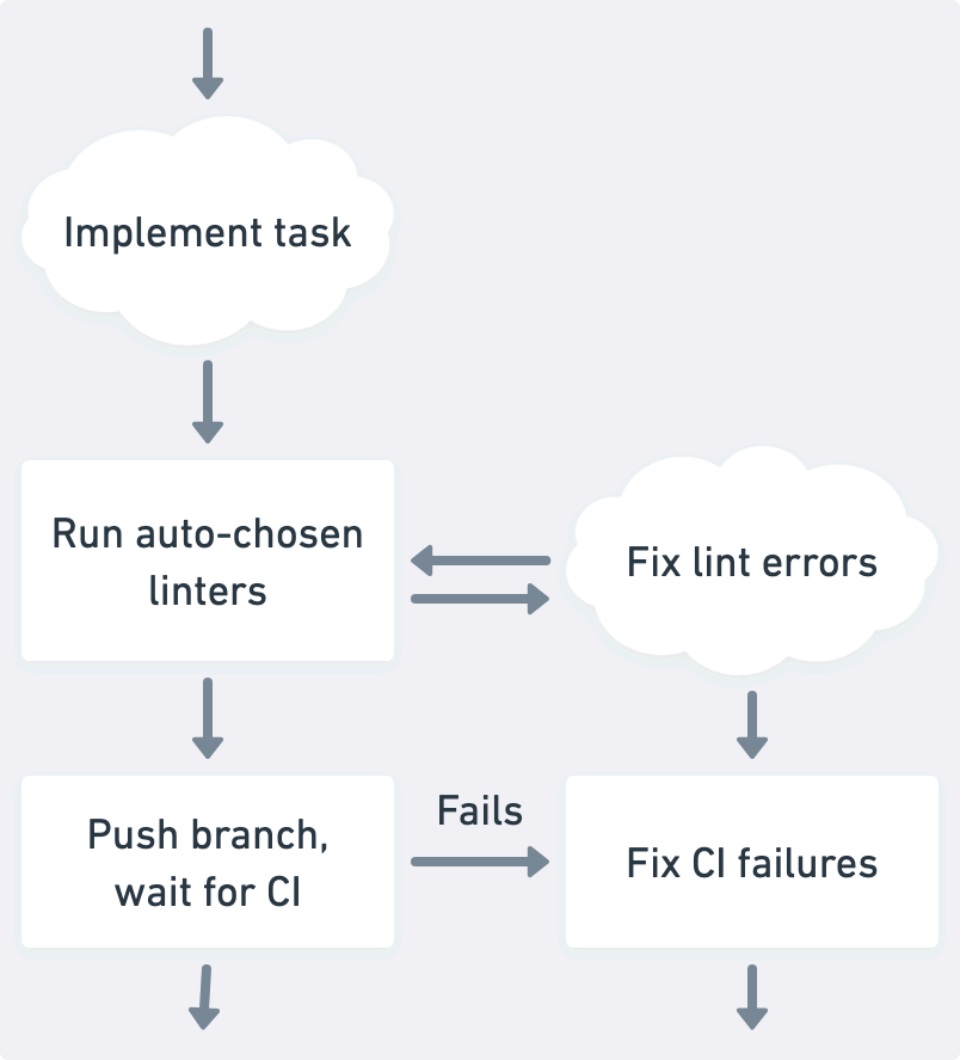
2.1 STRIPE: MINIONS

Scale: 1,300+ agent-generated pull requests merged per week as of February 2026 (up from 1,000 at the time of the Part 1 blog post). All are human-reviewed.

Agent harness: A fork of Block's open-source Goose agent, customised for Stripe's environment. Stripe evaluated off-the-shelf coding agents and chose to build on an open-source base rather than use tools like Claude Code or Cursor Agent because of the unattended execution requirement. Most coding agents are optimised for human supervision.

Execution environment: Stripe's existing devbox infrastructure. A devbox is an AWS EC2 instance pre-warmed with Stripe's full codebase and running services, originally built for human developers. They spin up in under 10 seconds. Devboxes run in Stripe's QA environment with no production access, no access to real user data, and no arbitrary network egress. The security properties were a consequence of existing infrastructure design, not something built specifically for agents.

Orchestration (blueprints): Stripe uses a hybrid orchestration pattern they call blueprints. A blueprint is a state machine that interleaves deterministic code nodes with agentic nodes. Deterministic nodes always execute the same way (run configured linters, push changes, trigger CI). Agentic nodes use an LLM loop for tasks requiring judgment (implement the task, fix CI failures). The LLM is invoked only where deterministic code is insufficient. This produces reliability at scale: small decisions that can be handled deterministically are not left to the LLM.



Stripe's blueprint orchestration pattern. Source: Stripe Engineering Blog

Tooling: Stripe built a centralised internal MCP server called Toolshed with approximately 500 tools. Each agent run is configured with a curated subset of around 15 tools relevant to its task. Providing the full tool set degrades performance through context window saturation.

CI policy: Maximum two rounds of CI iteration per agent run. If the agent cannot pass CI in two attempts, the branch is handed to a human engineer. No infinite retry loops.

FAILURE MODES ADDRESSED

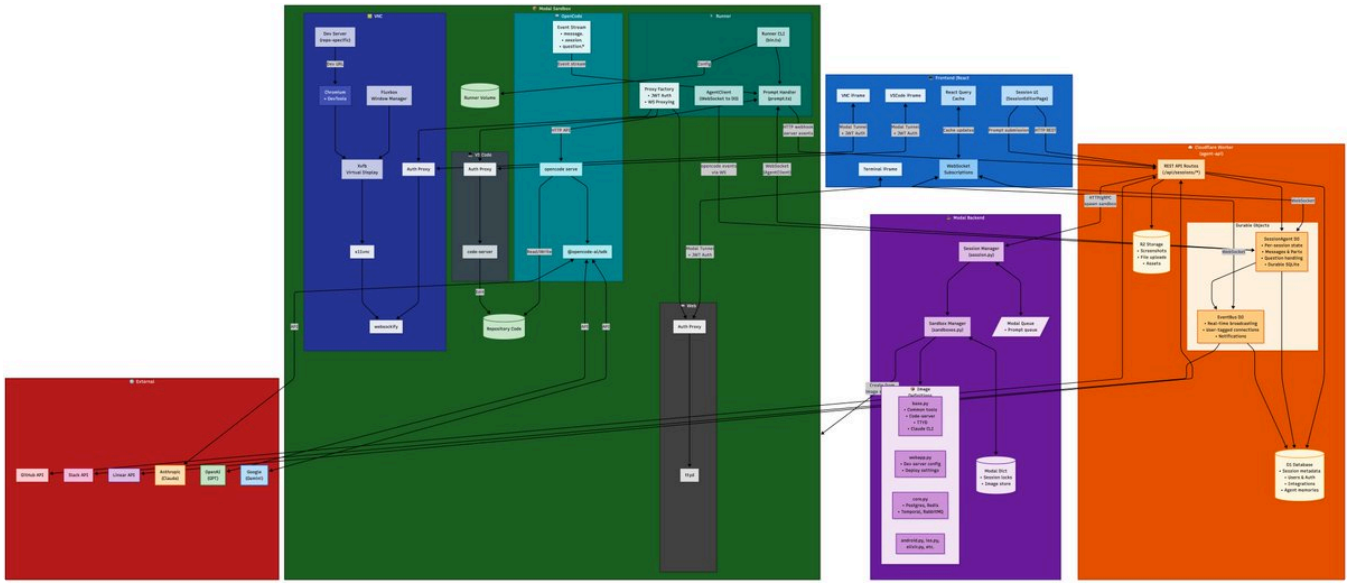
Context window saturation. Stripe discovered that providing all 500 Toolshed tools degraded agent performance. Curating to ~15 tools per task was the fix. More context is not better context.

CI retry loops. Without explicit bounds, agents retry failing CI indefinitely, consuming compute and producing increasingly divergent code. Stripe's two-round cap is a design

decision, not a limitation. Unbounded retry loops are the most common source of wasted compute in early agent deployments.

Source: Stripe Engineering Blog, "Minions" Parts 1 and 2 (2026)

2.2 RAMP: INSPECT



Ramp's Inspect architecture. Source: Rahul Pandey (@rahulgs)

Scale: As of February 2026, Inspect accounts for over half of all merged pull requests across Ramp's frontend and backend repositories. Over 80% of Inspect itself is written by Inspect.

Agent harness: OpenCode, running inside Modal Sandboxes. Ramp did not build a custom agent or custom execution infrastructure. The custom work concentrated on integrations with Ramp's internal tooling (Sentry, Datadog, LaunchDarkly, Temporal), orchestration logic, and client interfaces.

Execution environment: Modal Sandboxes. Each session contains a full-stack environment: Postgres, Redis, Temporal, RabbitMQ, plus a VS Code server, web terminal, and a VNC stack with Chromium for browser-based visual verification. The agent can take before-and-after screenshots and navigate the application in a real browser. All services run inside the sandbox at local-process latency, no network hop between agent and test suite.

State management: A Modal cron job clones each repository, installs dependencies, and saves a filesystem snapshot every 30 minutes. New sessions start from the latest snapshot, reducing startup time to seconds.

Clients: Engineers interact with Inspect via Slack, a web interface with an embedded VS Code editor, or a Chrome extension. The Chrome extension allows non-engineers to select UI elements directly for modification. Product managers and designers use Inspect without engineering involvement.

Adoption approach: Inspect was not mandated. It was deployed and adoption followed from product quality. Ramp's engineering team has noted that agents which are slower or less capable than local alternatives will not be adopted regardless of governance features.

> FAILURE MODES ADDRESSED
>
> **Credential scope creep.** The path of least resistance is giving agents the same credentials as the developer who configured them. Ramp's sandbox architecture enforces scoped, per-run credentials with least-privilege access. Without this, security teams will correctly veto autonomous agents.
>
> **Sandbox escape and network egress.** Agents in environments with unrestricted network access can exfiltrate code or credentials. Agents with no network access cannot reach internal systems. Ramp's Modal Sandboxes provide scoped network access: connections to internal services are allowed, arbitrary outbound traffic is blocked.

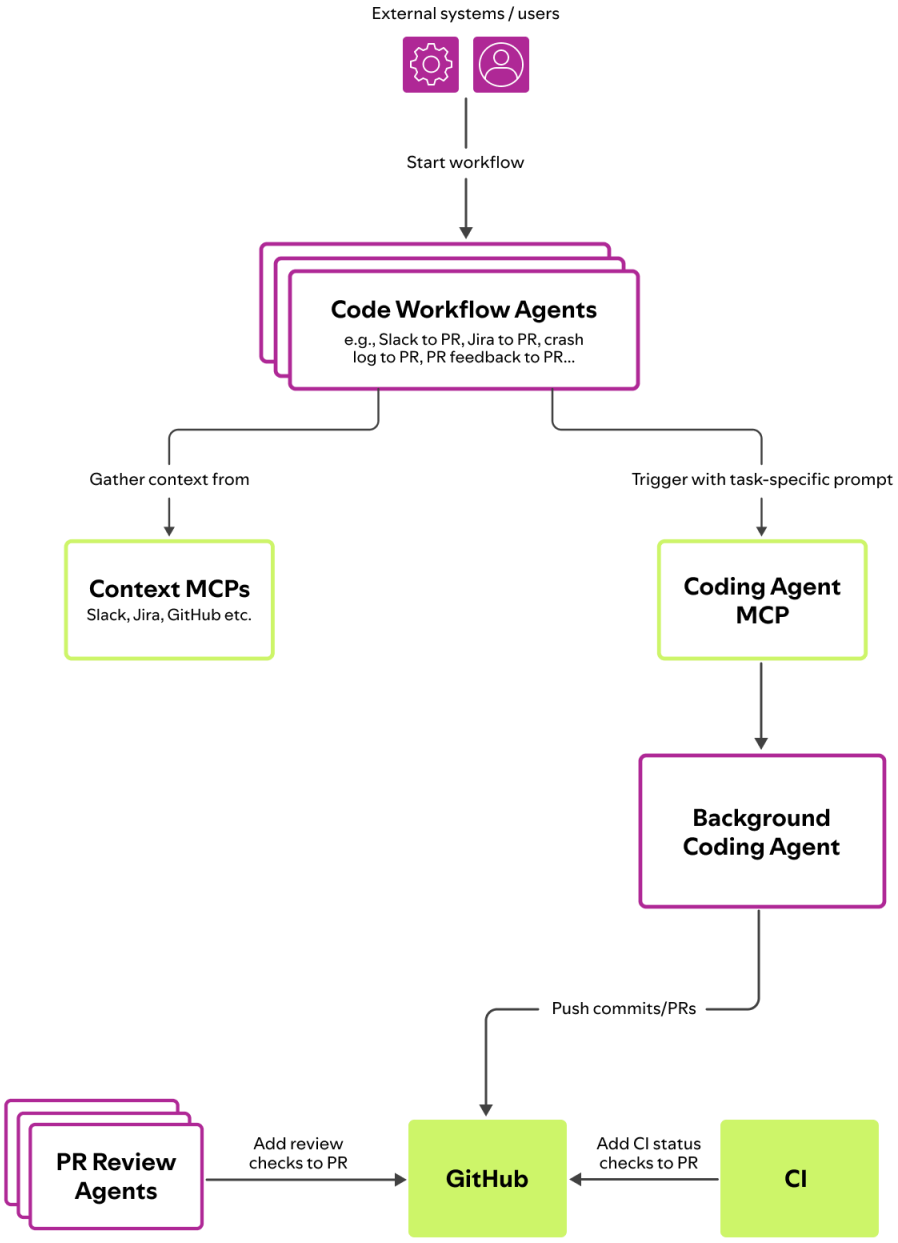
Source: Ramp Builders Blog (2025) and Modal Engineering Blog (February 2026)

2.3 SPOTIFY: HONK

Scale: 1,500+ agent-generated pull requests as of November 2025. Since mid-2024, around half of Spotify's pull requests have been automated by their Fleet Management system. On the Q4 2025 earnings call (February 10, 2026), Co-CEO Gustav Söderström stated: "my most senior engineers... say that they haven't written a single line of code since December."

Agent harness: Claude Code, integrated with an internal system called Honk. Mobile-first workflow: engineers send prompts from Slack, receive build status in Slack, and merge from their phones before arriving at the office. The iOS build pipeline is wired into the agent workflow, allowing mobile engineers to ship without opening Xcode.

Fleet Management foundation: Honk builds on Spotify's existing Fleet Management platform, a framework for applying code transformations across all repositories. The agent replaced deterministic migration scripts with prompt-driven code transformations, while the surrounding infrastructure (targeting repos, opening PRs, getting reviews, merging) stayed the same. A custom internal CLI delegates prompt execution to agents, runs formatting and linting via MCP, evaluates diffs using LLMs as a judge, and captures traces in MLflow.



Spotify's agent architecture. Source: Spotify Engineering Blog

Context engineering: Spotify's most significant infrastructure investment was in context: wiring organisational knowledge into agent workflows. This includes style guides, architecture decision records, internal API documentation, and patterns extracted from historical code reviews. Spotify distinguishes between a generic agent (one that produces technically correct output) and a Honk-configured agent (one that produces output consistent with Spotify's codebase conventions). Both use the same underlying model. The context produces the difference.

Feedback loops: Spotify measured agent output quality as a continuous metric and ran systematic evaluation of agent runs to feed improvements back into context and

configuration. Quality was not treated as a binary pass/fail threshold at launch.

Rollout: Progressive. Low-risk, high-volume tasks were automated first. Scope expanded as confidence grew. The team tracked adoption metrics and quality metrics in parallel, expanding only when both trended in the right direction.

> FAILURE MODE ADDRESSED
>
> **Review fatigue.** When agent output volume exceeds review capacity, either review quality drops (rubber-stamping) or the queue backs up and throughput gains disappear. Spotify's progressive rollout mitigates this: scale agent output to match review capacity, not the other way around. Start with tasks where review is lightweight (dependency updates, lint fixes) and expand as the team builds review habits for agent-generated code.

Source: Spotify Engineering Blog, three-part series (November 2025)

COMMON PATTERNS

There are a few consistencies across all three deployments:

- All three run agents inside full development environments, not code-execution-only sandboxes
- All three had cloud development infrastructure in place before layering agents on top
- All three invested heavily in context engineering (wiring organisational knowledge into agent workflows)
- All three enforce human review gates on all agent output
- All three started with bounded, low-risk tasks and expanded scope progressively

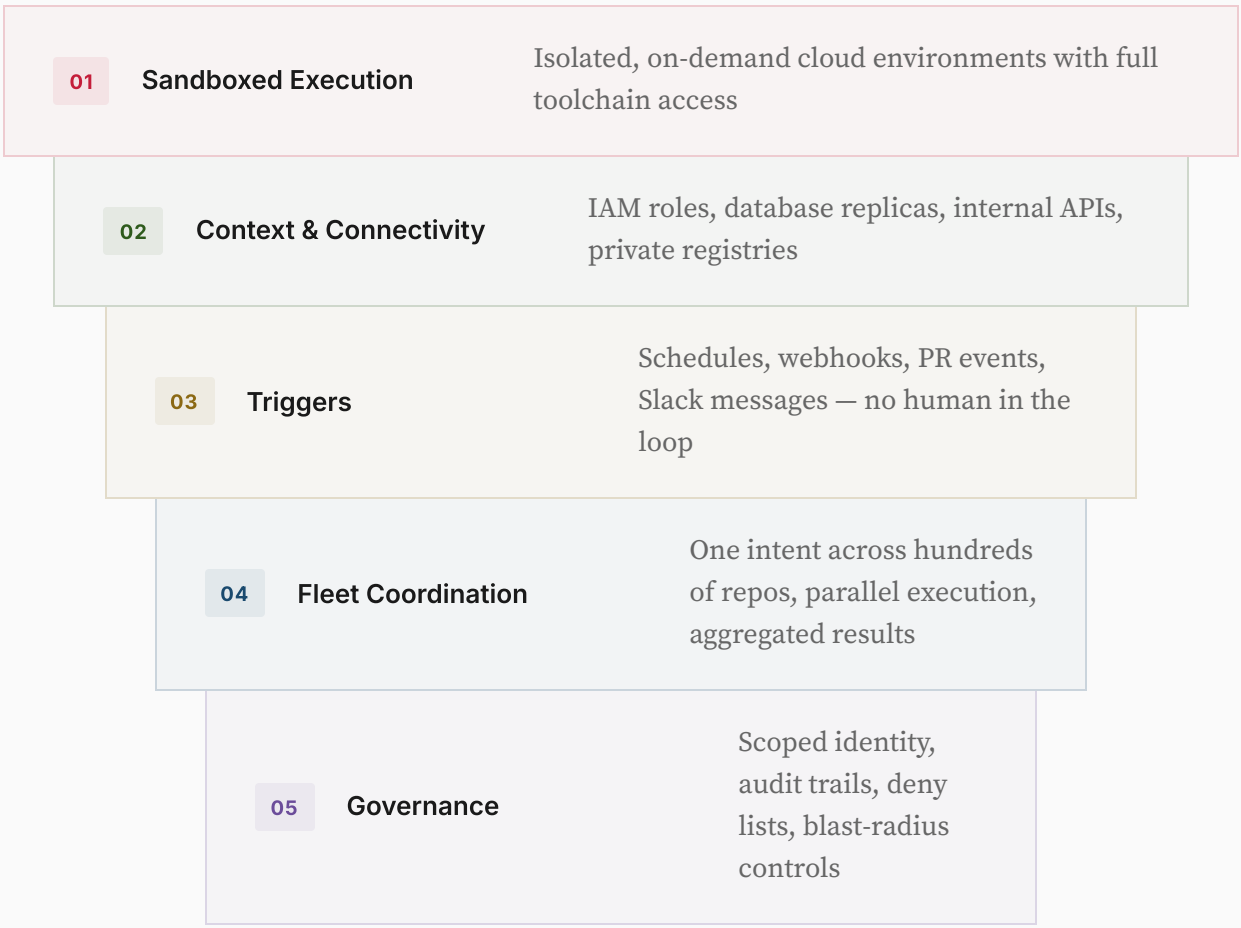
---

3. Infrastructure Primitives

Running background agents in an enterprise environment requires five infrastructure primitives. Each one is necessary. The absence of any single primitive either blocks

adoption or creates security and reliability problems that typically result in security team vetoes.

INFRASTRUCTURE PRIMITIVES



Each layer builds on the one above. Full details for each primitive follow below.

3.1 SANDBOXED EXECUTION ENVIRONMENTS

Each agent run requires an isolated execution environment (a VM or container) that includes a complete development toolchain: the same compilers, linters, test runners, build systems, databases, and internal services the developer would have locally. The environment should be reproducible and disposable. Hundreds of environments should be able to run concurrently without interference.

Why this is required: Agents that cannot reproduce the developer's local environment produce output that requires manual rework before it can be reviewed or merged.

Agents that share state across runs produce unpredictable results. Isolation is the prerequisite for reliability at fleet scale.

Pattern distinction: There are two common patterns for execution isolation. In the first, the agent runs inside a full development environment (a VM with the codebase, test suite, databases, and internal services). In the second, the agent runs on a server and calls out to a remote sandbox API for code execution only. The first pattern produces more capable agents. The second is more common in consumer-facing agent products. All three enterprise deployments covered in this paper (Stripe, Ramp, Spotify) use the first pattern.

SANDBOX ARCHITECTURE PATTERNS

PATTERN 1

Agent has dev environment

The agent has a full development environment. A VM running a dev container with your codebase, test suite, databases, and internal network access.

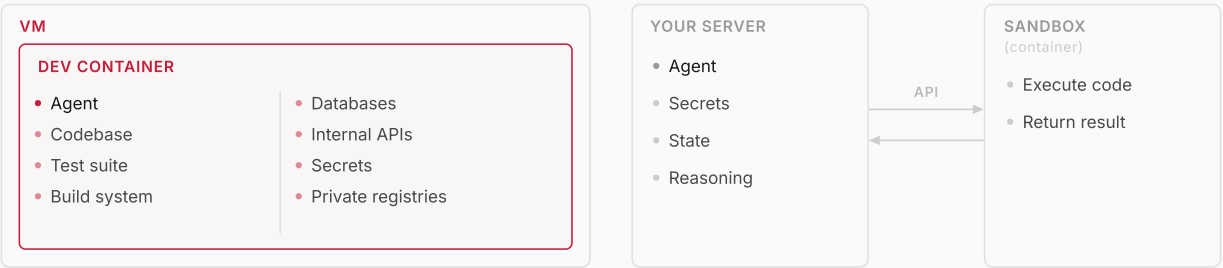
This is the closest to how a human developer works. Every enterprise that has shared their agent architecture publicly including Stripe, Ramp, and Spotify chose this pattern.

PATTERN 2

Sandbox as tool

The agent runs on a server or locally. When it needs to execute code, it calls a separate remote sandbox via API. The sandbox runs the code and returns the result. This keeps secrets and execution somewhat isolated, but the agent can only execute code and not fully develop.

Best suited for companies building agent products than organizations improving their own engineering workflows.



3.2 CONTEXT AND CONNECTIVITY

Agents operating on enterprise codebases need access to the internal systems those codebases depend on: internal package registries, database replicas, internal APIs, IAM roles, observability tooling. A sandboxed environment that cannot reach these systems will produce code that fails to build or test against the real codebase.

Context also encompasses organisational knowledge: coding standards, architecture decision records, patterns from past reviews, style guides, internal library documentation. This information shapes the difference between code that is technically correct and code that fits the codebase. Stripe, Ramp, and Spotify all invested significantly in this layer.

3.3 TRIGGERS

If every agent run requires a developer to type a prompt, the workflow has not been automated, only the work itself has. Triggers connect agents to the events that make automation possible:

- **Scheduled.** Run on a timer. Dependency updates nightly, lint enforcement weekly, documentation freshness checks daily.
- **Event-driven.** Respond to system signals. A CVE is published, a PR is opened, an alert fires, a CI run fails.
- **Fleet.** One trigger dispatches one agent per repository, all running in parallel.
- **Swarm.** One trigger dispatches multiple agents working on different aspects of a single task, with results converging.

3.4 FLEET COORDINATION

Fleet coordination is the capability that separates background agents from background coding sessions. One task description triggers parallel agent runs, one per repository, each working independently. Progress is tracked centrally. Results are aggregated. Failed runs are retried or escalated.

Updating one repository is a task for a coding agent. Updating 500 repositories in response to a CVE disclosure is a fleet task. Without fleet coordination, each repository

requires a separate human invocation.

3.5 GOVERNANCE

Agents are actors in the system and require the same access controls applied to human contributors: identity, scoped permissions, and audit trails.

Governance enforced by system prompt instruction ("do not modify production data") is a request that the LLM may or may not honour under edge cases. Governance enforced at the execution layer (deny lists, scoped credentials, deterministic command blocking, network egress restrictions) is structural and does not depend on LLM compliance.

The practical requirements: each agent run should operate under a scoped identity with least-privilege credentials. Human review gates should sit between agent output and any environment with production access. Audit logs should capture what each agent run did, what it attempted, and what it produced.

Why this is required: Security teams will, correctly, reject autonomous agents that run with a developer's full credentials and no audit trail. Governance at the execution layer is typically what unlocks security team approval.

---

4. Ona Internal Data

Background agents were deployed internally at Ona starting December 18, 2025. The data below covers September 2025 through March 2026, providing a before-and-after comparison with the same team, the same codebase, and the same product roadmap.

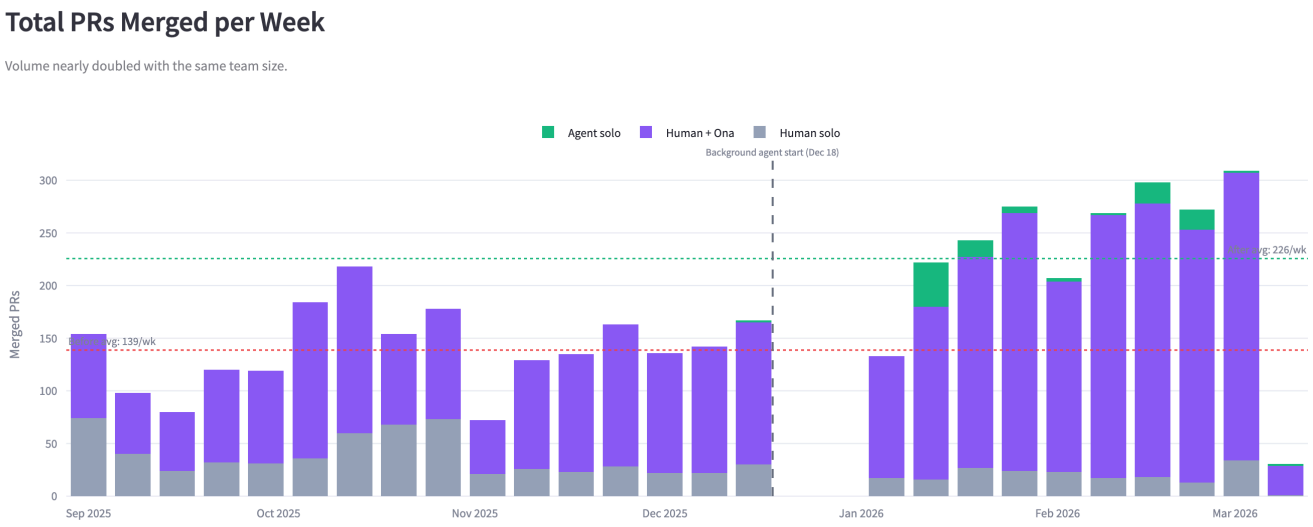
4.1 METHODOLOGY

All data is sourced from Git commit and pull request metadata across Ona's production repositories. PRs are classified into three categories based on author and tooling signals:

- **Human solo:** PRs authored entirely by a human developer without agent assistance
- **Human + Ona:** PRs where a human developer worked with Ona's agent tooling
- **Agent solo:** PRs authored entirely by a background agent, triggered by an event, with no human involvement until review

Team size remained constant across the measurement period. No engineers were added or removed. The product roadmap did not change in scope or priority. The "before" and "after" periods reflect the same team doing the same work with different tooling.

4.2 TOTAL PRS MERGED PER WEEK

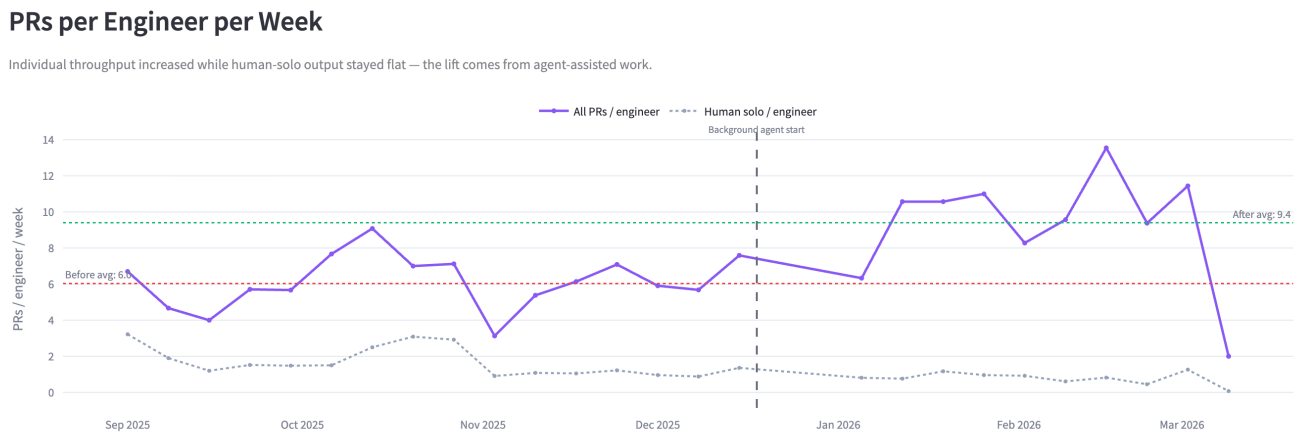


Volume nearly doubled with the same team size. The green segments (agent solo) appear after the December 18 deployment date. Human solo output (grey) decreased as engineers shifted to reviewing and orchestrating agent work.

Before background agents: 139 merged PRs per week on average. After: 226 per week. A 63% increase with no change in team size.

The composition shifted. Human solo PRs decreased as engineers moved toward reviewing and orchestrating agent work rather than writing every change by hand. Human + Ona PRs (where engineers worked alongside agent tooling) became the dominant category. Agent solo PRs (fully autonomous, event-triggered) appeared after the December 18 deployment and grew steadily.

4.3 PRS PER ENGINEER PER WEEK



Individual throughput increased while human-solo output stayed flat. The lift comes from agent-assisted work.

Before: 6.0 PRs per engineer per week. After: 9.4. A 57% increase in individual throughput.

The human solo line (dotted) stayed flat or declined slightly. Engineers did not start writing more code. They started reviewing and directing more agent-generated work. The productivity gain came entirely from the agent-assisted and agent-solo categories.

This is the pattern the constraint analysis in the executive summary predicts. The bottleneck shifted from "writing code" to "reviewing and merging code," and background agents operating on downstream tasks (test generation, dependency updates, lint fixes) kept the pipeline moving.

---

5. The Build-vs-Buy Reality

Every organisation in the case studies above built their own background agent infrastructure. They had to. When they started, nothing existed to buy.

Look at what it cost them.

Stripe built a dedicated internal team ("Leverage"), custom cloud infrastructure on AWS, a centralised MCP server with 500 tools, and deep integrations with their internal build

and deployment systems. This sat on top of devbox infrastructure they had been investing in for years before agents were possible.

Ramp assembled an applied AI team and built custom integrations with Sentry, Datadog, LaunchDarkly, and Temporal on top of Modal's sandbox infrastructure. Even with Modal handling the execution layer, the custom work was substantial.

Spotify invested years in Fleet Management before agents were even possible, then built a custom CLI, GCP integrations, MLflow tracing, and a multi-agent architecture on top.

These are companies with thousands of engineers, dedicated platform teams, and the organisational patience to invest in infrastructure that takes quarters to pay off. They did not build background agent platforms as a side project. They built them because developer infrastructure is their competitive advantage.

Do you have what they had?

A dedicated platform engineering team with spare capacity. Deep expertise in sandboxed execution environments. An existing fleet management system to build on top of. The organisational patience to invest in infrastructure that will not show ROI for two or three quarters. The engineering talent to build, maintain, and evolve a custom agent platform while the underlying models and tools change every month.

Most organisations do not. Building background agent infrastructure is a fundamentally different kind of investment than adopting a coding agent. Adopting Copilot is a procurement decision. Building a background agent platform is a multi-quarter, multi-team engineering programme.

The companies that succeeded built on infrastructure they had been investing in for years. Spotify's Fleet Management system predated their agent work. Stripe's devbox infrastructure predated Minions. Ramp's SDLC tooling predated Inspect. The agents did not create the infrastructure. They inherited it.

If you have not already made those investments, you are not starting from where Stripe started. You are starting from scratch. And starting from scratch on infrastructure that the best-resourced engineering teams in the world spent years building is not a realistic path for most organisations.

THE ALTERNATIVE

Buy the primitives (sandboxed execution, governance, context and connectivity, triggers, fleet coordination) from a platform that has already built them. Then focus your engineering talent on the work that is actually unique to your organisation: defining the workflows, tuning the agents, and building the feedback loops that make the output fit your codebase.

The companies in this paper built because they had no choice. You have a choice.

What you do with it will determine whether your organisation is running background agents in six months, or still running coding agents on laptops and wondering why cycle time has not moved.

---

6. The Self-Driving Codebase

We opened this paper with a vision: a codebase that drives itself. Stripe, Ramp, and Spotify showed that it is not a vision. It is an engineering programme with specific infrastructure requirements and measurable outcomes.

The shift is from imperative to declarative. You stop telling agents what to do and start declaring what should be true:

- *Dependency versions should always be current.*
- *Test coverage should never drop below the threshold.*
- *Security vulnerabilities should be patched within 24 hours.*
- *Every PR should be reviewed before a human sees it.*

You declare the desired state. Agents ensure it. Continuously, across every repo, without being asked. This is the same transition that happened with infrastructure-as-code, GitOps, and policy-as-code, applied to the entire software delivery lifecycle. Engineers who have lived through those transitions will recognise the pattern immediately.

The organisations that get there will not be the ones with the best coding agents. They will be the ones that built, or bought, the infrastructure to make their codebases self-driving.

Start building

Ona provides the infrastructure primitives covered in this paper: sandboxed execution environments, governance, context and connectivity, triggers, and fleet coordination. If you are evaluating background agents for your organisation, we can help you get from Phase 2 to Phase 3.

Talk to Ona

Further Reading

1. 1,500+ PRs Later: Spotify's Journey with Our Background Coding Agent — Spotify
2. Background Coding Agents: Context Engineering — Spotify
3. Background Coding Agents: Predictable Results Through Strong Feedback Loops — Spotify
4. Why We Built Our Background Agent — Ramp
5. Minions: Stripe's One-Shot, End-to-End Coding Agents — Stripe
6. Minions: Stripe's One-Shot, End-to-End Coding Agents, Part 2 — Stripe
7. Towards Self-Driving Codebases — Cursor
8. Expanding Our Long-Running Agents Research Preview — Cursor
9. Time Between Disengagements: The Rise of the Software Conductor — Ona
10. Industrializing Software Development — Ona