Background Agents and the Self-Driving Codebase

An enterprise guide to the next era of software delivery

Executive Summary

Coding agents made individual developers faster. Background agents make codebases self-driving — autonomous, event-triggered, running across the software delivery lifecycle without a human at the keyboard.

Most organisations adopted coding agents expecting organisational transformation. What they got was individual acceleration: engineers writing code faster, PRs arriving sooner, backlogs growing instead of shrinking. Cycle time didn't budge. DORA metrics stayed flat. The bottleneck moved, but nobody noticed — because the gains felt real at the individual level.

This paper lays out the maturity model that separates organisations still optimising individual stations from those redesigning the entire production system. It covers:

- **Where the industry is** — the coding agent wave, the productivity paradox, and the localhost ceiling
- **The false summit** — why running multiple agents in parallel feels like transformation but isn't
- **What background agents actually are** — and how they differ from coding agents and CI/CD pipelines
- **The infrastructure primitives** — sandboxed execution, governance, context and connectivity, triggers, and fleet coordination
- **A three-step playbook** — from establishing primitives to finding bottlenecks to scaling the software factory
- **Case study teardowns** — how Stripe, Ramp, and Spotify built their background agent infrastructure

The audience is engineering leaders at organisations with 50 to 5,000 engineers who have deployed coding agents and are asking: *what's next?*

---

1. The State of AI in Software Development

THE CODING AGENT WAVE

The adoption curve for AI coding tools has been the fastest in the history of developer tooling. GitHub Copilot crossed 1.8 million paid subscribers within two years of launch. Cursor, Claude Code, and Windsurf followed — each offering progressively more autonomous capabilities, from autocomplete to multi-step task execution.

These tools work. Engineers report writing code faster, spending less time on boilerplate, and moving through unfamiliar codebases with less friction. The individual productivity gains are real and measurable.

But individual productivity gains are not the same as organisational transformation.

THE PRODUCTIVITY PARADOX

Here is the pattern playing out across hundreds of engineering organisations right now:

1. The team adopts coding agents. Engineers are faster. PRs arrive sooner.
2. The review queue backs up. Reviewers can't keep pace with the increased volume.
3. CI pipelines become the bottleneck. More PRs means more builds, more flaky tests, more failures to investigate.
4. Merge conflicts multiply. More concurrent work on the same codebase means more collisions.
5. Cycle time stays flat — or gets worse. The backlog grows. DORA metrics don't move.

The gains are compounding with the individual, not the organisation. The longer you invest in coding agents without addressing the system around them, the deeper you entrench.

This is the productivity paradox of AI in software development: **individual speed does not equal organisational velocity.**

THE LOCALHOST CEILING

Coding agents run on your machine. They use your CPU, your memory, your network connection, your credentials. This creates a hard ceiling:

- **Resource contention.** Multiple agents fighting over the same machine state — ports, file locks, environment variables.
- **Security exposure.** Agents running with your full credentials, accessing everything you can access, with no isolation between sessions.
- **Availability.** Everything stops when the laptop sleeps, the VPN drops, or you close the lid.
- **Scale.** Your machine has 8 cores. Your organisation has 500 repositories.

Engineers have already figured out that agents should run somewhere else. They're buying Mac Minis, setting up git worktrees, running multiple terminal sessions. These are all the same signal: the current model is untenable for professional engineering at scale.

The instinct is correct. The solution is wrong.

---

2. The False Summit

You're running five Claude Code sessions in parallel. You've got git worktrees set up so agents don't collide. You bought a Mac Mini to run agents overnight. You're using background mode in your IDE. It feels like you've arrived.

You haven't. This is the false summit.

WHAT THE FALSE SUMMIT LOOKS LIKE

- Multiple coding agents running in parallel on a developer's machine or dedicated hardware

- Git worktrees to avoid branch conflicts between concurrent agent sessions
- Spare machines (Mac Minis, cloud VMs) repurposed as always-on agent runners
- Background mode in IDEs — kick off a task, check back later
- A growing sense that "we're already doing background agents"

WHY IT STALLS

Parallel agents on your machine are still:

- **Human-triggered.** Every agent run starts with a developer typing a prompt. You haven't automated the workflow — just the work.
- **Single-repo scoped.** Each agent works on one repository at a time. Updating a dependency across 200 repos means 200 manual invocations.
- **Ungoverned.** No audit trail, no permissions model, no blast-radius controls. The agent has whatever access you have.
- **Uncoordinated.** No fleet management, no progress tracking, no aggregated results. Each agent is an island.
- **Fragile.** The machine sleeps, the session dies, the work is lost.

The gap between "I use agents in the background" and "my organisation runs background agents across the SDLC autonomously" is where most organisations stall.

THE BOTTLENECK MOVED

The theory of constraints teaches that improving a non-bottleneck is waste. Code generation was a genuine bottleneck — developers spent significant time writing boilerplate, navigating unfamiliar code, and translating intent into syntax. Coding agents solved that bottleneck.

Now the bottleneck is everything else: code review, testing, CI pipeline maintenance, dependency management, security patching, merge conflict resolution, release coordination. These are the stations on the assembly line that coding agents don't touch — and they're where cycle time actually lives.

Speeding up code generation while leaving these bottlenecks untouched is like installing a faster engine in a car stuck in traffic. The constraint isn't the engine. It's the road.

3. What Is a Background Agent?

A coding agent needs your machine and your attention. A background agent needs neither.

It runs in its own development environment in the cloud: full toolchain, test suite, secrets access, network connectivity — everything. Completely decoupled from your device and your session.

Kick one off from your laptop, check the result from your phone. Trigger it from a PR, a Slack thread, a Linear ticket, a webhook, or a cron schedule. You're not steering it. You're not watching it. It's an asynchronous task: delegate, walk away, review later.

CODING AGENTS VS. BACKGROUND AGENTS

Dimension	*Coding agent*	*Background agent*
Where they run	Your laptop or local machine	Cloud infrastructure, triggered remotely
How they're triggered	You invoke them manually	Events, schedules, Slack messages, API calls — any signal
Scope	Single task in one repo	Across repos, teams, and the full SDLC
Developer's role	In the loop — watching, steering, iterating	On the loop — prompt, walk away, review the output
Session lifetime	Tied to your machine and your session	Runs independently for as long as it needs
Governance	Inherits your credentials and access	Scoped permissions, audit trails, blast-radius controls

Dimension	*Coding agent*	*Background agent*
Coordination	One agent, one task	Fleets across hundreds of repos, swarms on complex tasks

BACKGROUND AGENTS VS. CI/CD PIPELINES

CI/CD pipelines execute predefined steps — build, test, deploy. They don't make decisions or generate new code. Background agents are autonomous: they receive a trigger, reason about the problem, write code, run tests, and open a pull request.

A CI pipeline tells you a dependency is outdated. A background agent updates it, verifies nothing breaks, and submits the fix for review.

A CI pipeline reports a test failure. A background agent investigates the failure, identifies the root cause, writes a fix, and opens a PR — before a developer is paged.

THE TRIGGER TAXONOMY

Background agents respond to different types of signals, each mapping to a different scope and cadence:

Scheduled agents. Triggered on a timer. Predictable, bounded, high-volume. Dependency updates every night, lint sweeps every week, coverage enforcement every sprint.

Event-driven agents. Triggered by system events — a PR opened, a CVE published, an alert fired. Reactive, concurrent, always listening. The agent responds to the event, not to a human.

Agent fleets. One task across many repositories. Each agent works independently in its own environment and produces its own contribution. Update a configuration across 500 repos in parallel.

Agent swarms. Many agents, one outcome. Every agent works on a different facet of a complex task and results converge into a single deliverable. Decompose a large migration into parallel workstreams.

ON THE LOOP, NOT IN THE LOOP

The shift from coding agents to background agents changes the developer's role. Instead of writing every line, developers move "on the loop" — reviewing agent output, calibrating behaviour, designing systems, and focusing on work that requires judgment.

You still look at the code. You still own the decisions. The difference is you're reviewing work, not doing it keystroke by keystroke.

---

4. The Infrastructure Primitives

Autonomous agents need infrastructure that doesn't exist on your laptop. The building blocks below are what separate a demo from a deployment. Each one unlocks the next.

PRIMITIVE 1: SANDBOXED EXECUTION ENVIRONMENTS

What it is. Agents running in the background need their own execution environment with a full toolchain, the ability to run tests, and access to secrets and systems. Environments should be isolated, reproducible, and have close parity to production systems.

Why it matters. Without isolation, one agent's failure can cascade into another's work. Without a full toolchain, the agent can't run the same commands your engineers run. Without reproducibility, you can't trust the output.

What happens without it. Agents share machine state, fight over ports and file locks, and produce results that work on one machine but not another. Security teams veto autonomous agents because there's no blast-radius control.

What it looks like in practice. Each agent gets its own development environment — a full VM or container with the complete toolchain, test suite, and build system. The environment spins up on demand, runs the task, and tears down when done. Hundreds can run in parallel without interference.

The architecture question. There are two patterns emerging: centralised (all agent environments run in a shared cloud platform) and distributed (agent environments run inside the customer's own infrastructure). The choice depends on security requirements, data residency, and how much of the stack you want to own.

PRIMITIVE 2: GOVERNANCE

What it is. Agents are actors in your system. They need the same controls as human contributors — identity, permissions, audit trails. The difference: governance enforced by a system prompt ("please don't delete files") is a suggestion. Governance enforced at the execution layer — deny lists, scoped credentials, deterministic command blocking — is actual governance.

Why it matters. Without governance, security teams veto autonomous agents entirely. And they're right to. An ungoverned agent with broad credentials is a liability, not an asset.

What happens without it. No audit trail for what agents did and why. No way to scope access per task. No way to enforce policies consistently across agent runs. Every agent has the same access as the developer who triggered it — which is usually far more access than the task requires.

What it looks like in practice. Each agent run has a scoped identity with least-privilege credentials. Command deny lists prevent destructive operations. Every action is logged to an audit trail. Human review gates (typically pull requests) sit between agent output and production. Blast-radius controls ensure a single failure can't cascade.

PRIMITIVE 3: CONTEXT AND CONNECTIVITY

What it is. A sandbox that can't reach your internal systems is a toy. Agents need to assume IAM roles, query database replicas, hit internal APIs, and pull from private registries — all from inside your network.

Why it matters. Enterprise codebases don't exist in isolation. They depend on internal services, private packages, proprietary build tools, and data that lives behind the firewall. An agent that can't reach these systems can't do real work.

What happens without it. Agents produce code that compiles but doesn't integrate. They can't run integration tests, can't access the real dependency graph, can't validate against production-like data. The output requires manual rework to actually ship.

What it looks like in practice. Agent environments run inside the customer's network or have secure tunnels to internal systems. They can assume IAM roles, access private registries, query database replicas, and use internal CLIs — the same access a developer's machine has, but scoped and audited.

PRIMITIVE 4: TRIGGERS AND AUTOMATION

What it is. If every agent run starts with a developer typing a prompt, you haven't automated the workflow — just the work. Triggers connect agents to the events that matter: schedules, webhooks,

system signals, conversational interfaces.

Why it matters. Triggers are what make background agents *background*. Without them, you have coding agents that happen to run on remote machines. With them, you have autonomous workflows that respond to the system, not to humans.

What happens without it. Every automation requires a human to remember to invoke it. The value is limited to the tasks someone thinks to trigger. The system is reactive to humans, not to events.

What it looks like in practice. A PR is opened — an agent reviews it before a human sees it. A CVE is published — an agent patches every affected repo within hours. A cron job fires at midnight — an agent sweeps every repo for outdated dependencies. A developer sends a Slack message — a fleet of agents fans out across the codebase.

PRIMITIVE 5: FLEET COORDINATION

What it is. Updating one repository is a coding agent task. Updating 500 is a fleet task. Fleet coordination means the same sandbox, replicated across every repository that needs the change — parallel provisioning, progress tracking, aggregated results.

Why it matters. This is where individual productivity becomes organisational throughput. A single agent updating a single repo is useful. A fleet updating every repo in the organisation simultaneously is transformative.

What happens without it. Large-scale changes require manual invocation per repo, manual tracking of progress, and manual aggregation of results. The "500 repo update" becomes a multi-week project instead of a multi-hour operation.

What it looks like in practice. One intent ("update this dependency across all repos") triggers parallel agent runs — one per repo. Each agent works independently in its own environment. Progress is tracked centrally. Results are aggregated into a dashboard. Failed runs are retried or escalated automatically.

---

5. The Playbook: Three Steps to Organisational Transformation

STEP 1: ESTABLISH THE PRIMITIVES

You can't run background agents without the infrastructure to support them. The five primitives — sandboxed execution, governance, context and connectivity, triggers, and fleet coordination — are the foundation.

The build-vs-buy decision matters here. Building your own sandbox infrastructure is a multi-quarter investment that requires dedicated platform engineering resources. The companies that have done it (Stripe, Ramp, Shopify) had the scale and the engineering talent to justify it. Most organisations don't — and shouldn't.

What to do first: - Evaluate whether your current infrastructure can support isolated, on-demand agent environments - Assess your governance requirements — what do your security and compliance teams need before they'll approve autonomous agents? - Map your connectivity requirements — what internal systems do agents need to reach? - Start with manual triggers and add automation as you build confidence

STEP 2: FIND YOUR BOTTLENECKS

The primitives give you the capability. Where you apply them is what matters. That means doing the work: surveying your developers, sitting with your teams, mapping where time goes. Every organisation's bottlenecks are different. The ones worth solving first aren't always obvious.

Seven high-value starting points:

1. **Code reviews that pile up faster than ever.** PRs sit for hours while reviewers context-switch. A background agent reviews every PR before a human sees it, so reviewers focus on design, not formatting.

2. **CI failures nobody has time to investigate.** Your build breaks and you spend 20 minutes reading logs to find the one flaky test. A background agent triages failures, identifies flaky tests, and opens fix PRs.

3. **Merge conflicts that pile up between branches.** You pull main and spend 30 minutes resolving conflicts. A background agent resolves the mechanical conflicts and flags the rest.

4. **CVEs that sit unpatched for weeks.** A GitHub advisory arrives and joins the backlog. A background agent applies the fix, runs tests, and opens PRs across every affected repo.

5. **Test coverage that never gets written.** Legacy code ships with zero coverage because there's no time. A background agent generates tests for uncovered paths and validates them against existing behaviour.

6. **Code standards that vary between every repo.** Enforcement is patchy across teams. A background agent checks every PR against your standards and auto-fixes the straightforward violations.

7. **Release notes that nobody writes.** Someone spends half a day reading merged PRs. A background agent compiles changes, groups them by type, and drafts notes ready for review.

The principle: Start with the most boring work, not the most impressive. The highest-value automation targets are repetitive, well-defined tasks with bounded blast radius. Dependency updates. Lint enforcement. Test generation. These are the tasks that cost the most aggregate time and carry the least risk if an agent gets them wrong.

STEP 3: SCALE YOUR SOFTWARE FACTORY

The engineering organisation is an industrial system. Today, developers stand at every station: writing, reviewing, testing, deploying. Background agents change the operating model. The factory runs, but your engineers move *on* the loop instead of *in* it.

The checklist — what "done" looks like:

- ☐ Every PR is reviewed by an agent before a human sees it
- ☐ CI failures are investigated and fixed before a developer is paged
- ☐ Developers never manually resolve merge conflicts on agent PRs
- ☐ Agents do the first-pass investigation into every production incident
- ☐ Security vulnerabilities are patched within hours, not sprints

This isn't a destination you reach in a quarter. It's an operating model you build toward incrementally — one automated workflow at a time, each one freeing engineering time for the work that requires human judgment.

6. Case Study Teardowns

STRIPE — MINIONS

Stripe's Minions platform is one of the most publicly documented background agent deployments. Built on custom infrastructure inside Stripe's network, Minions are one-shot, end-to-end coding agents that receive a task, execute it autonomously, and produce a pull request.

Key architectural decisions: - **Custom agent framework** on top of foundation models with multiple model support and task-specific agent profiles - **Full development environments** on internal cloud infrastructure — each agent gets an isolated VM with the complete Stripe toolchain - **Internal system access** — agents run inside Stripe's network with access to internal build systems, test runners, linters, and deployment tools - **Human review gates** — all agent output goes through the standard PR review process

What's transferable: The pattern of giving agents full development environments (not just code generation sandboxes) with access to the real toolchain. The insistence on human review gates for all output. The investment in task-specific agent profiles rather than one-size-fits-all prompting.

RAMP

Ramp published a detailed account of why they built their own background agent infrastructure rather than using off-the-shelf tools. Their reasoning centred on the gap between what coding agents could do and what their engineering organisation needed.

Key architectural decisions: - **Purpose-built for their SDLC** — the agent infrastructure was designed around Ramp's specific workflows, not generic coding tasks - **Event-driven triggers** — agents respond to system events (PRs, deployments, alerts) rather than human prompts - **Integrated with internal systems** — agents have access to Ramp's internal APIs, databases, and tooling

What's transferable: The decision framework for build-vs-buy. The recognition that generic coding agents don't address organisational bottlenecks. The focus on event-driven triggers as the key differentiator from "coding agents running in the background."

SPOTIFY

Spotify's engineering team published a three-part series documenting their background coding agent deployment — over 1,500 PRs generated by agents running autonomously.

Key architectural decisions: - **Context engineering** — extensive investment in wiring organisational knowledge (style guides, ADRs, internal APIs, past reviews) into agent workflows - **Strong feedback loops** — systematic evaluation of agent output quality, with metrics driving continuous improvement of agent behaviour - **Progressive rollout** — started with low-risk, high-volume tasks and expanded scope as confidence grew

What's transferable: The emphasis on context engineering as the differentiator between generic output and output that fits your codebase. The feedback loop methodology — treating agent quality as a metric to optimise, not a binary pass/fail. The progressive rollout strategy that builds organisational trust incrementally.

---

7. Frequently Asked Questions

Can I just run coding agents in the background on my laptop?

You can, and many teams do — multiple terminals, git worktrees, maybe a Mac Mini humming in the corner. But that's agents running in the background, not background agents. There's no event triggering, no governance, no audit trail, and everything stops when the machine sleeps. The gap between those two things is where most organisations stall.

What infrastructure do background agents need?

Three things that don't exist on your laptop: isolated compute environments that spin up on demand, an event system that routes triggers (PR opened, CVE published, Slack message, cron schedule) to the right agent, and a governance layer for permissions, audit trails, and blast-radius controls.

Are background agents safe?

They're as safe as the infrastructure you put around them. The key controls are sandboxed environments with scoped access, human review gates on all output (typically via pull requests), audit

trails for every action, and bounded blast radius so a single failure can't cascade. The risk isn't the agent — it's running agents without these controls.

Do background agents replace developers?

No. They shift what developers do. Instead of writing every line, developers move "on the loop" — reviewing agent output, calibrating behaviour, designing systems, and focusing on work that requires judgment. You still look at the code. You still own the decisions. The difference is you're reviewing work, not doing it keystroke by keystroke.

What are common use cases for background agents?

The highest-value starting points are repetitive, well-defined tasks with bounded blast radius: dependency updates across hundreds of repos, CVE remediation within hours of disclosure, CI pipeline migrations, lint and standards enforcement, test coverage expansion, and code review triage. Start with the workflows costing the most time, money, or risk.

How are background agents different from CI/CD pipelines?

CI/CD pipelines execute predefined steps — build, test, deploy. They don't make decisions or generate new code. Background agents are autonomous: they receive a trigger, reason about the problem, write code, run tests, and open a pull request. A CI pipeline tells you a dependency is outdated. A background agent updates it, verifies nothing breaks, and submits the fix for review.

How do we measure ROI on background agent infrastructure?

Start with time-to-resolution metrics on the tasks you automate. If CVE patching took two weeks and now takes two hours, that's measurable. If code review turnaround dropped from 8 hours to 20 minutes for the first pass, that's measurable. Aggregate across the number of repos and the frequency of the task. The ROI compounds with scale — updating one repo saves minutes, updating 500 saves weeks.

What's the change management path for engineering teams?

Start with the most boring work. Engineers won't resist automating dependency updates or lint fixes — they'll be relieved. Build trust with low-risk, high-volume tasks before expanding to more complex workflows. Expect that developers will over-review agent output for the first two weeks, then under-review it. Build review habits early. The teams that succeed treat agent output like junior developer output: review everything, trust incrementally.

8. Towards a Self-Driving Codebase

The factory floor is running. Code is being written, reviewed, tested, deployed — continuously, autonomously. Your engineers are observing. Setting constraints. Verifying outcomes.

This is the destination: a codebase that drives itself.

Not a codebase without humans — a codebase where humans do different work. Instead of writing every line, they're designing agent workflows. Instead of reviewing every PR manually, they're setting the constraints that agents enforce. Instead of investigating every CI failure, they're improving the system that investigates for them.

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. The engineers who've lived through those transitions will recognise the pattern immediately.

The organisations that get there won't be the ones with the best coding agents. They'll be the ones that redesigned the system around autonomous execution — the ones that stopped optimising individual stations and started building the factory.

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