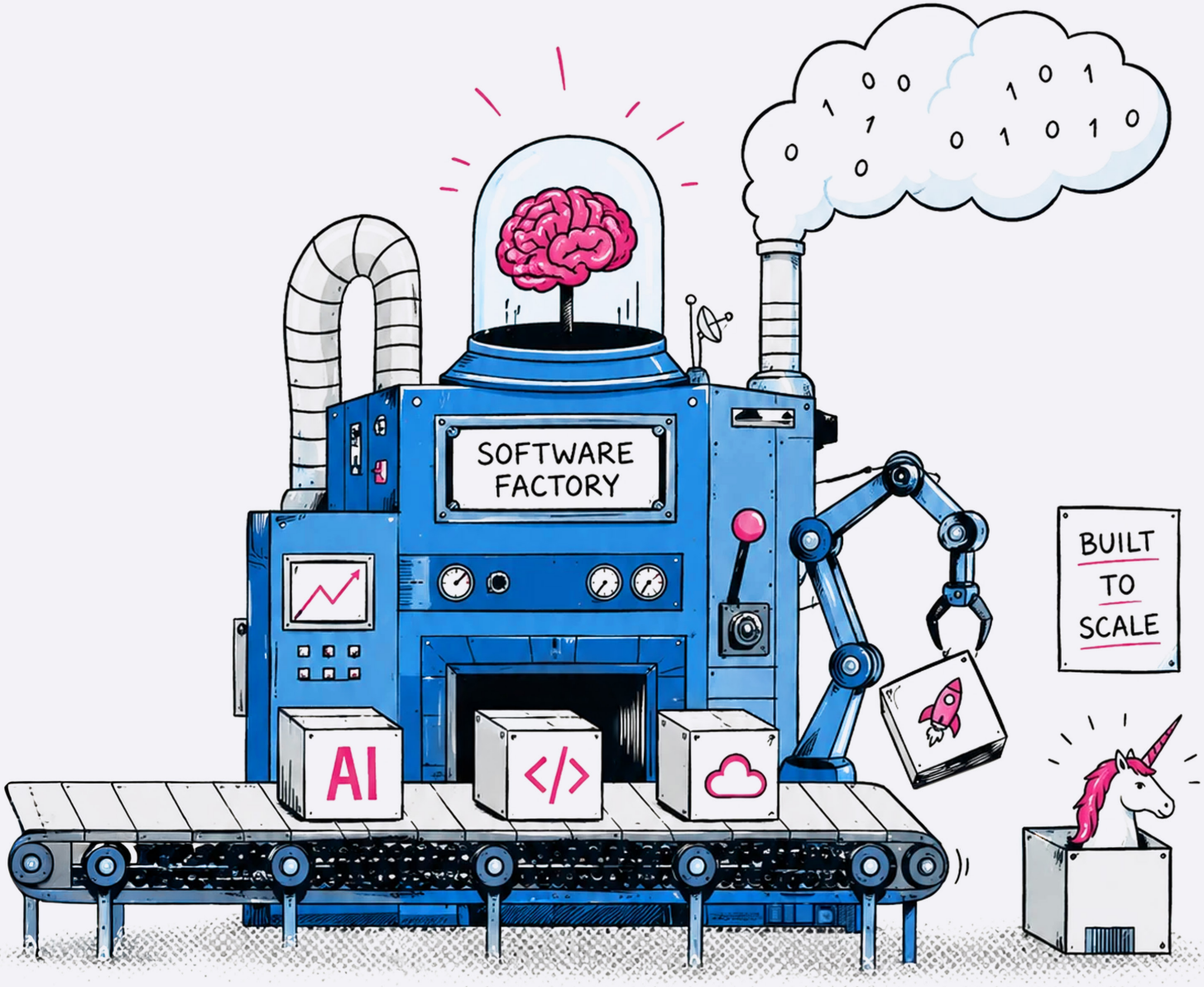
# Software Factories at Enterprise Scale

## A Federated Platform for Agentic Development



# Software Factories at Enterprise Scale

## A Federated Platform for Agentic Development

A handful of engineering organisations have shown, in 2026, what agentic software development looks like at scale. Stripe merges around 1,300 agent -written pull requests a week. A small team at OpenAI built close to a million lines of agent-written code in five months, with no human reviewing before merge. Ramp's agents handle about half its pull requests. These are production systems, and the throughput comes from the system around the model more than from the model itself.

It is tempting to read those stories as a template, and that is where the trouble starts. Every one of them runs in conditions most companies don't share. Stripe's agents sit on top of a single monorepo with roughly three million tests and around 500 internal tools that engineers built over half a decade. OpenAI's team had one codebase and one toolchain to themselves. The factory works because the infrastructure underneath it was put in place, slowly, long before the agents arrived.

Most enterprises are not built that way. A company of any size has several product lines, codebases at very different stages of maturity, regulated and unregulated work, teams spread across countries, and businesses that joined through acquisition and still run on their own conventions. The development organisation is less like one well-tended monorepo and more like a collection of small, independent ones, each with its own constraints. Drop a single central factory on top of that and most teams will find it doesn't fit. Ask every team to build their own and the organisation pays for the same platform many times over, while governance fragments and what one team learns rarely reaches the next.

This paper describes a third way: a federated platform that gives every team the capabilities the case studies describe, while letting each one keep the autonomy its codebase and risk profile require. The chapters that follow set out why the obvious paths fail, how a three-tier architecture divides ownership between the platform, the domain, and the team, why MCP is the seam that holds it together, and how the same platform can run unattended factories and human-in-the-loop assistants side by side. The aim throughout is to be honest about what is solved, what is not, and where to start.

# Contents

# The Case Studies and the Gap they Leave

The most discussed examples of agentic software development in 2026 are single-org, single-stack, single-domain. Stripe runs one Minions fleet against one Stripe monorepo and merges around 1,300 pull requests a week. OpenAI's harness team built what they call a “dark factory” against one codebase: three to seven engineers, roughly a million lines of agent-written code in five months, no human reviewer pre-merge. Ramp's Inspect handles around half the pull requests across its frontend and backend repos. Cognition's Devin runs inside customer environments, but each deployment is single-team. GitHub's Copilot Coding Agent works general-purpose at scale, but at the cost of being narrow per task.

The architecture underneath these is real. Each implementation integrates orchestration, isolated environments, structured context, validation gates, governance, and learning loops into one coordinated system. The pull request is the unit of work. The throughput numbers come from the system, not from the model. The pattern is the software factory.

The pattern is also narrow. Stripe's monorepo has roughly three million tests and around 500 internal MCP tools that human engineers built over half a decade. OpenAI's harness team had a single codebase and toolchain to themselves. These are the conditions under which those factories work. Most enterprise development organisations are not built that way.

A large company typically has multiple product lines, multiple codebases at different stages of maturity, teams in different countries, regulated and unregulated parts of the estate, and recent acquisitions still operating the way they did before. Even a single-product company at meaningful scale has platform teams, product teams, data teams, ML teams, and security teams whose demands on the engineering platform are genuinely different. The development organisation is not one Stripe. It is dozens of small Stripes, each with its own constraints, conventions, and risk profile.

When the case studies are the template, an enterprise dev org has a difficult choice. Build one central factory and roll it out, and watch most teams reject it because the platform doesn't fit their codebase, their stack, or their risk profile. Let every team build their own factory, and pay the cost in duplicated platform investment, fragmented governance, and lessons that never propagate. Neither answer is workable at enterprise scale.

This paper argues for a third option. A federated platform: shared infrastructure where it pays, local autonomy where it matters, the capabilities the case studies describe available to every team within the boundary conditions each team's reality demands.

*The Development Organisation is Not One Stripe. It is Dozens of Small Stripes.*

# Why the Obvious Paths Fail?

## TWO PATHS FAIL PREDICTABLY

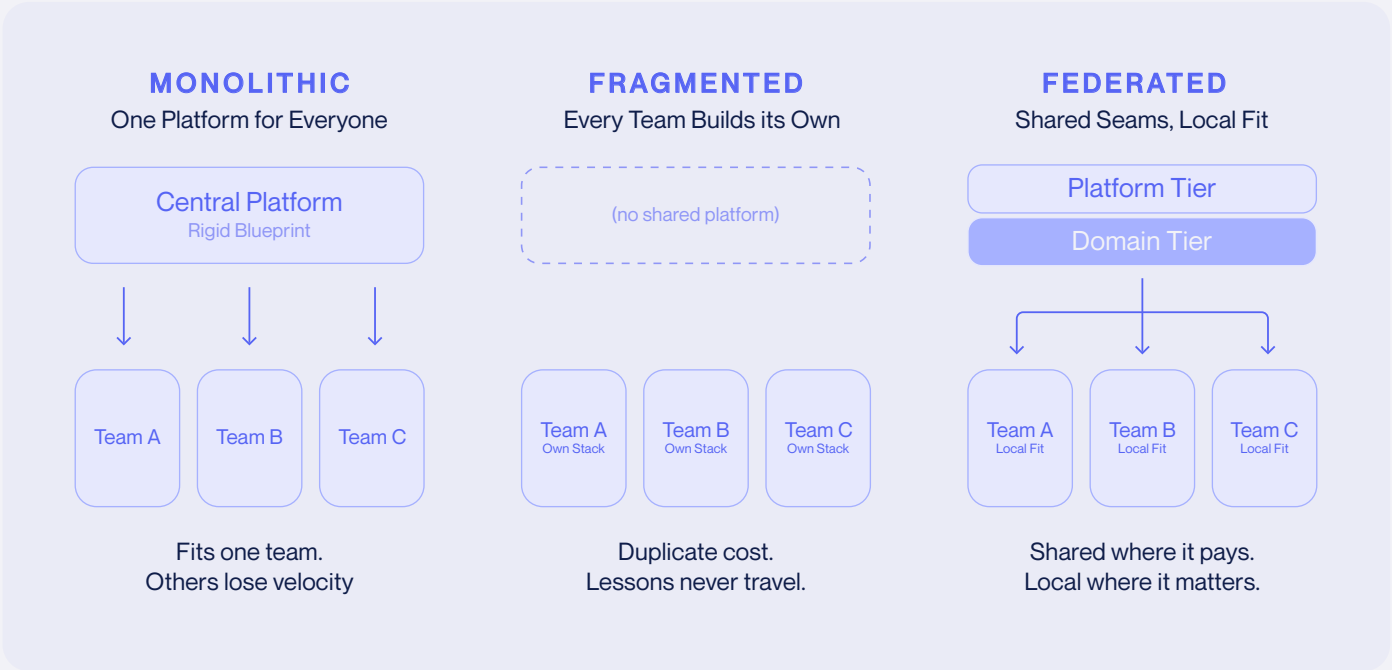
The first is the central platform team that builds one factory for the whole organisation. This works for the team that built it. For most other teams it doesn't fit: their codebase has different conventions, their test coverage is at a different maturity, their risk profile is different, their language and framework stack is different. Those teams either reject the platform (the central team's strategy becomes shelfware) or grudgingly adopt it while losing velocity to the impedance mismatch. In either case, the central team takes the blame.

This failure is structural, not a question of execution quality. A single platform that fits a brand-new core in TypeScript with a modern test suite cannot also fit a fifteen-year-old Java monolith with sparse coverage. Trying to make it fit produces something that works poorly for both.

The second failed path is the opposite. Every team builds its own agent harness against its own codebase. There is no shared identity directory, no shared MCP registry, no shared audit log, no shared blueprint library. The maintenance cost is paid many times over. Security and compliance fragment. The lessons learned by one team never reach the others. The whole reason to be one organisation disappears.

The middle path is layered federation. Some things owned centrally. Some things owned by clusters of related teams. Some things owned locally. Each layer has clear ownership and a clear contract with the layers above and below it.

## THE THREE APPROACHES (MONOLITHIC / FRAGMENTED / FEDERATED)



# The Three-tier Architecture

The Platform tier holds the central control plane. It includes the versioned blueprint registry, the scoped tool router that decides which MCP tools each agent role can reach, the identity directory that authenticates every user and every agent, and the immutable audit log. The Platform tier is owned by a central platform engineering team. It runs once.

A secure API proxy sits between the Platform tier and everything below it. The proxy holds real credentials. Agents never see them. When an agent in a team needs to call an MCP tool, the request goes through the proxy, which injects scoped tokens at egress and writes the call to the audit log on the way back. Each agent has its own non-human identity, distinct from any human user's, and the audit log preserves both identities for every action: which human delegated the work, and which agent executed it. Without that distinction, every action shows up in the log as either the proxy or the delegating human, and accountability collapses at agent volume. On-device agent registries that scrub credentials and manage tool access at the OS boundary have begun shipping with major desktop operating systems. The enterprise version runs as a multi-tenant cloud gateway rather than a per-machine registry, but the design principle is the same: credentials are scrubbed at the boundary, the boundary is the audit point, and the audit captures both the human and the non-human identity behind each action.

The Domain tier holds shared context across clusters of teams with related concerns. A domain can be defined in several ways depending on the organisation. It can be a regulatory domain (all teams handling payment data share PCI -compliant blueprints and the controls that go with them). It can be a technical domain (all teams on the new core stack share its conventions, tooling, and component library). It can be a business domain (all teams in the consumer business share customer-data patterns and conventions). It can be a geographic domain (all EU teams share GDPR-compliant patterns). Some teams sit in multiple domains; the Domain tier is plural by construction.

## *Blueprint once. Execute everywhere*

The Team tier is where execution happens. Each team runs its own isolated sandboxes and its own MCP servers, with local conventions on top. The runtime is chosen by what the work actually does, not by how heavy it feels. Any task that touches a filesystem, runs a shell, installs packages, or compiles code has to run inside a sandbox with a kernel underneath it: a container, a microVM, or a userspace-kernel sandbox. Lightweight edge isolates are a different category. They boot in milliseconds and cost a small fraction of a full container, but they have no filesystem and no shell, so they fit stateless data transformation and API routing rather than coding work. Interactive work runs in a local devbox tied to the developer's own machine. The platform supports all of these because the blueprint runtime is decoupled from the execution layer.

## THE THREE-TIER ARCHITECTURE (PLATFORM / DOMAIN / TEAM)

Shared where it pays.
Local where it matters.

# The Six Capabilities a Software Factory Needs

Six capabilities show up in every working software factory. Orchestration coordinates the work of agents across tasks. Isolated environments give each agent a clean place to run without conflicts. Context and memory keep the agent aware of the codebase and conventions it needs to work within. Validation checks output before it ships. Governance keeps the system from doing things it shouldn't. Learning closes the loop from production back into the harness.

At enterprise scale, the question for each capability is where in the federation it lives.

Orchestration is split. Blueprints stay at Platform, versioned. Teams can fork them. Drift is observable: one team running a custom build-and-deploy blueprint while every other team in the same domain runs the canonical version shows up in the registry. The question of whether that fork should propagate upward or be retired can be answered from data rather than from opinion.

Isolated environments are local. The runtime is chosen by what the work actually does, not by how heavy it feels. Anything that touches a filesystem or runs a shell needs a real sandbox: a container, a microVM, or a userspace-kernel sandbox. Stripe's devbox model (pre-warmed with the source tree loaded and caches kept warm, ten-second spin-up) is a good template for greenfield code generation. Stateless data transformation and API routing can run in lightweight edge isolates that boot in milliseconds. Interactive work runs locally on the developer's machine. The federated platform supports all of these because the blueprint runtime is decoupled from the execution layer.

Context and memory is where the MCP server becomes the federation seam. The domain context MCP at the Team tier holds local operational knowledge: vendor SLAs, the conventions in this codebase, the recent decisions made in code review. The Domain tier MCP holds shared context across related teams. The Platform MCP holds the cross-cutting context: chart of accounts conventions, the architecture cookbook, the component registry. An agent serving a team pulls from all three. The user doesn't need to know which tier a given piece of context came from.

***Every software factory needs six capabilities:*** *orchestration, isolation, context, validation, governance, and learning.*

The operational knowledge in the Team MCP is what compounds over time. Generic engineering practice is being absorbed into foundation models quickly. The specific operational knowledge inside an organisation (the particular decisions this codebase has made, the vendor relationships this team relies on, the institutional memory of decisions made in thousands of design discussions) does not exist in any public corpus and never will. Encoding it into MCP servers is what builds a durable advantage.

Validation looks similar at every tier, but the specifics are local. Each team runs its own validation harness that inherits the Platform baseline and extends it for checks specific to its codebase or domain. A change to a payment system must pass PCI compliance verification before a human ever sees it. A change to a marketing site does not. The shared part is the harness pattern. The specific checks are local.

Governance is where central ownership pays the biggest dividend. The Platform gateway scrubs credentials, enforces RBAC, and writes the immutable audit log. All tool calls and data access flow through it. Model inference can happen at the edge or on a developer's local workstation; the governance point is the tool boundary, not the model call. The platform's guarantee is that no agent reaches a production system without the call being logged and authorised. The platform does not guarantee that every model token passes through head office, and it does not need to.

Learning propagates upward through the federation. Data does not. When one team learns that a particular code-review pattern catches 30% more bugs in production, the lesson can be anonymised and contributed to the Domain reference. The data stays in the originating team. Other teams in the domain can adopt the lesson without ever seeing the originating team's code. Standardisation happens through demonstrated value.

### CAPABILITIES-BY-TIER MATRIX

| CAPABILITY | PLATFORM | DOMAIN | TEAM |
|---|---|---|---|
| Orchestration | Blueprint registry | — | Fork, execute |
| Isolated environments | — | — | Sandbox chosen per workload |
| Context and memory | Cross-cutting | Cluster-shared | Operational |
| Validation | Baseline harness | Compliance checks | Local extensions |
| Governance | Identity and audit | — | Inherits tokens |
| Learning | Aggregates | Propagates | Generates |

# Two Patterns Running in Parallel

Software factory writing in 2026 has converged on the dark factory pattern: agents work unattended, humans review at the merge boundary or accept post-hoc classification of failures. Stripe, OpenAI's harness team, and Ramp all run versions of this. The pattern works.

It only works under specific preconditions. Stripe's Minions are reliable because the platform underneath them is mature: roughly three million tests, around 500 internal MCP tools, devboxes that human engineers have used for half a decade. The agents inherited an environment that was already in place. David Cramer at Sentry has been blunt about the alternative: “The patch generation is awful. I will be the first to tell you that it is not that good, and everybody's patch generation is awful.” His position is that this technology will be human-in-the-loop for a long time, and the value comes from finding bugs you would not have found yourself.

Both positions are correct, for different parts of the estate.

Greenfield work (new core systems, new product surfaces, modern stacks) usually has the preconditions: a current test suite, a standardised dev environment, conventions encoded in an AGENTS.md. The dark factory pattern fits.
Brownfield work (the legacy estate that came in through acquisition, the older internal systems that nobody has had time to modernise) usually does not. The test suite is thin, the conventions are in people's heads, the dev environment is bespoke. Running an unattended agent against that estate produces broken PRs at scale.
The honest design is two patterns running in parallel on the same platform. Dark-factory blueprints run against greenfield repos under the conditions that support them.
Interactive-assistant blueprints run against legacy repos where the human stays in the loop. The blueprint runtime is the same. The classification of which pattern applies to which repo is metadata in the blueprint registry. Promising dark-factory autonomy on a legacy estate is the most common way these initiatives lose credibility, and it is avoidable.

# MCP as the Federation Seam

The single most consequential architectural decision at enterprise scale is making MCP the standard interface between agents and every internal system.

Internal applications get wrapped as MCP servers. The accounting system, the customer database, the CRM, the regulatory filing system, the engineering deployment pipeline.

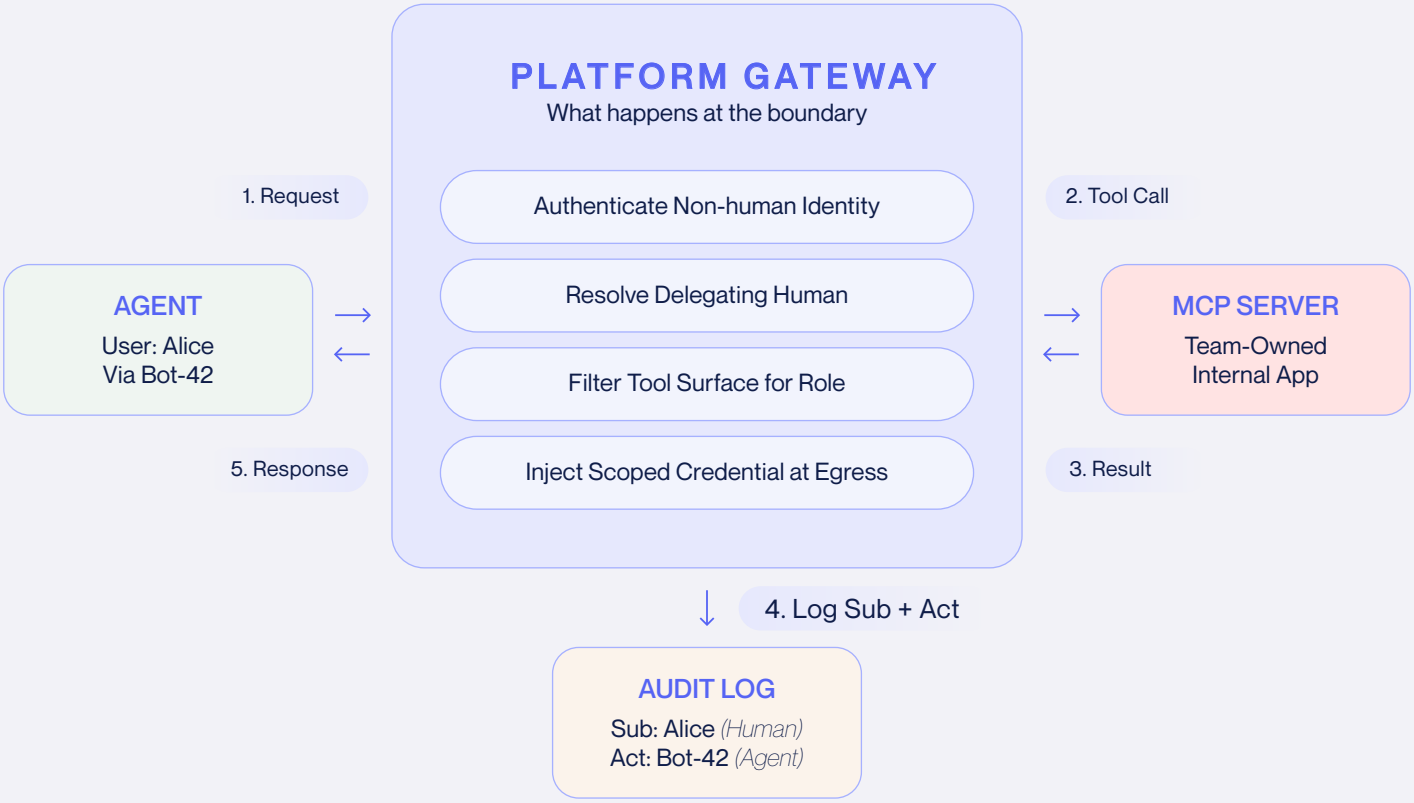
Each exposes a small, well-scoped set of tools. The team that owns the application owns the MCP wrapper. The central platform team does not become a bottleneck for new integrations.

A central MCP registry knows which servers exist, which team owns each one, and which tools they expose. The Platform gateway sits in front of every call. An agent never has direct access

to an MCP server; it asks the gateway, which checks RBAC, scrubs credentials, makes the call, and returns the result with the transaction logged. The gateway also filters the tool surface presented to each agent so that an agent which needs three tools does not see a catalogue of four hundred. Oversized tool catalogues degrade model reasoning and widen the attack surface. The NSA's May 2026 information sheet on MCP security calls this out directly, alongside concerns like unverified task propagation between MCP servers and session reuse through unscoped tokens. These are governance problems rather than protocol problems, and the gateway is where they get solved.

This is a substantial commitment. It means every team building an internal application accepts a contract: expose your service through MCP, conform to the schema, accept the auditing. The payoff is that every agent on the platform, in every team, can use the application immediately, with governance enforced by the path rather than by checklists. Without this commitment, the federated platform fragments back into point integrations and the network effect disappears.

The closest published precedents come from operating system vendors that have shipped on-device agent registries at the platform level. The cloud-multi-tenant version of the pattern has fewer published implementations. In our work across enterprise development organisations, this is the central technical bet of the next five years for any organisation that wants agents to operate across more than one system.

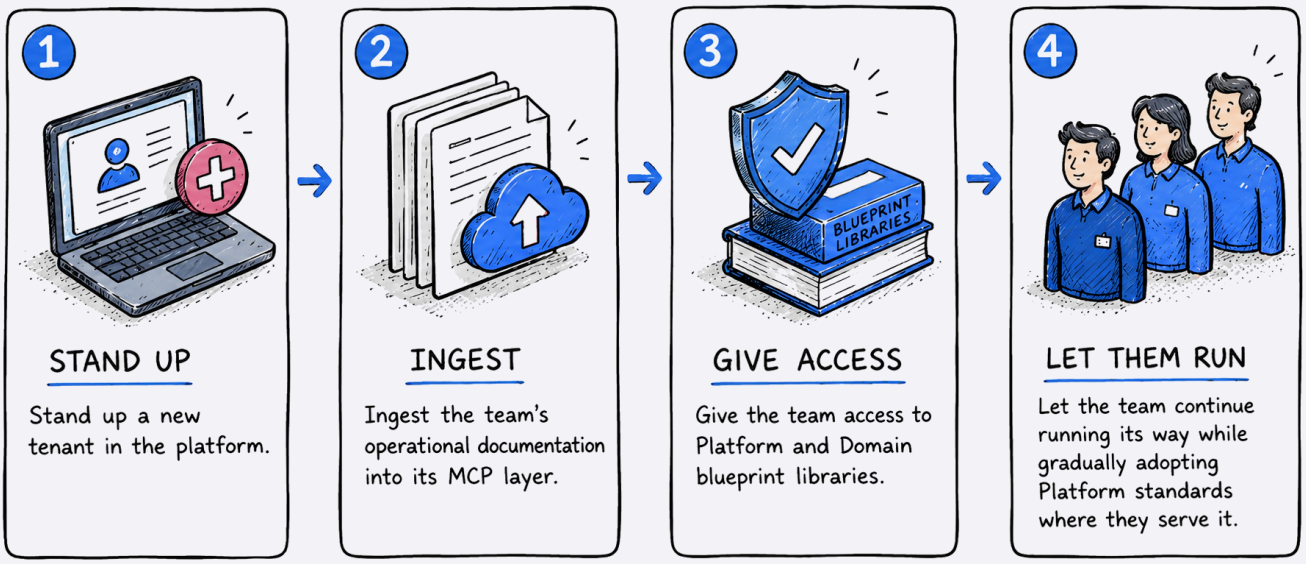


# Onboarding New Teams

When the platform exists, onboarding a new team becomes a procedural exercise.

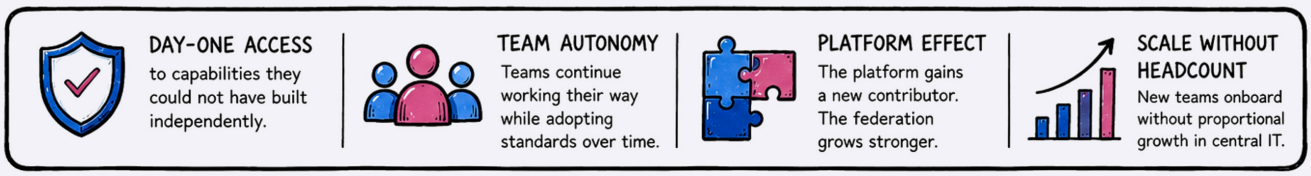
- Stand up a new tenant in the platform.
- Ingest the team's operational documentation into its MCP layer.
- Give the team access to Platform and Domain blueprint libraries.
- Let the team continue running its way while gradually adopting Platform standards where they serve it.

This applies to several distinct cases that look identical at the architectural level. A new internal team being spun up. A new product line getting its own engineering team. A regional office being established. An acquired business joining the group. All of them get the same onboarding flow.




The new team gets day-one access to capabilities it could not have built independently: coding agents, knowledge encoding, verification, the rest of the platform. The Platform tier gains a new contributor to the federation. The new team's operational uniqueness stays intact. Standardisation happens through demonstrated value.

The economics are most visible for acquisitions. Traditional integration of an acquired engineering organisation consumes 12 to 24 months of work and a substantial slice of central IT capacity. With a federated platform in place, the operational onboarding (the part that makes agents work against the acquired codebase) becomes a question of weeks. For internal expansions the absolute time saved is smaller, but the ratio is similar. The platform absorbs new teams without proportional growth in central headcount.



# Three Problems Still Open

The published case studies on software factories name five problems that remain unsolved even at single-team scale: memory and context at scale, evaluation versus reality, specification quality, governance velocity, and the talent pipeline. All five remain unsolved at enterprise scale. Three new problems appear on top of them.

The first is cross-team knowledge propagation without data leakage. A lesson learned in one team should reach another. The data behind the lesson should not. Today, the anonymisation and abstraction step is human-mediated and slow. Doing it automatically while preserving the lesson's specificity is an open problem. Active research on federated retrieval-augmented generation and privacy-preserving knowledge sharing points at the shape of a solution, but production - grade implementations at enterprise scale remain rare. This is also the problem that determines how much value federation actually delivers, because the alternative (each team learning independently) wastes the network effect that justifies the platform in the first place.

The second is the fork-versus-adopt question. When a team forks a Platform blueprint, when is that a sign of valid local adaptation, and when is it tech debt? Today the answer comes from judgement. There is no data-driven framework for it, and the cost of getting it wrong is years of unnecessary maintenance on parallel implementations. A workable heuristic probably involves measuring outcome differences (does the fork actually produce better results than the canonical blueprint?), but the measurement infrastructure for outcome-attributable blueprint comparison does not yet exist in any published implementation.

The third is identity at federation scale. A user inside a team acts within that team. Their agent inherits scoped tokens from the Platform gateway. The auth model is straightforward in principle: the agent inherits the user's permissions and nothing more. In practice, it gets complicated. What if the user works across two teams? What if a Domain blueprint needs to read data from three teams at once? What if a Platform user, an internal auditor for example, needs read access to a team's data without that team seeing the audit? The answer is built from federated identity primitives, scoped tokens, and policy as code. The implementation work is non-trivial and not yet documented in public engineering posts.

These three problems do not block adoption. They get more interesting as a federation matures, and they are worth being honest about from the start.

***Software Factories Solve Many Problems, but Three Critical Enterprise-Scale Challenges Remain Unsolved.***

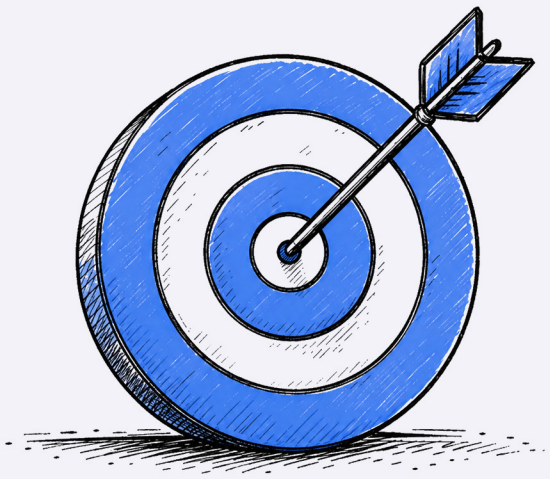
# What this Means for Engineering Leaders

Four principles hold across every team running a software factory in production. Use boring, well-documented technology because agents work more reliably against it. Keep context in the repo because anything outside the repo is invisible to the agent. Verify away from where you generate because agents will optimise for any signal you give them. Treat the agent platform as a product because the surrounding engineering practice shapes the outcome. At enterprise scale, add a fifth: the platform spans many teams but governs from one place.

The practical sequence is not to build the federated platform in one go. It is to build one factory against one team first, prove that it works, and design the Platform tier alongside it so that the second team joins as a tenant of an existing platform rather than as a copy of the first team's bespoke build. By the third team, the federation effect should be visible: shared blueprints adopted by demonstrated value, knowledge propagating across the Domain reference layer, the MCP registry growing as each team wraps its applications.

This is not the only way to build agents at scale, and it will not be the right shape for every organisation. For an enterprise development organisation that has teams with materially different demands, whether through organisational design, technical heritage, or M&A, and that needs to keep them autonomous while sharing the parts that should be shared, the federated platform is what makes the rest of the engineering work pay off.

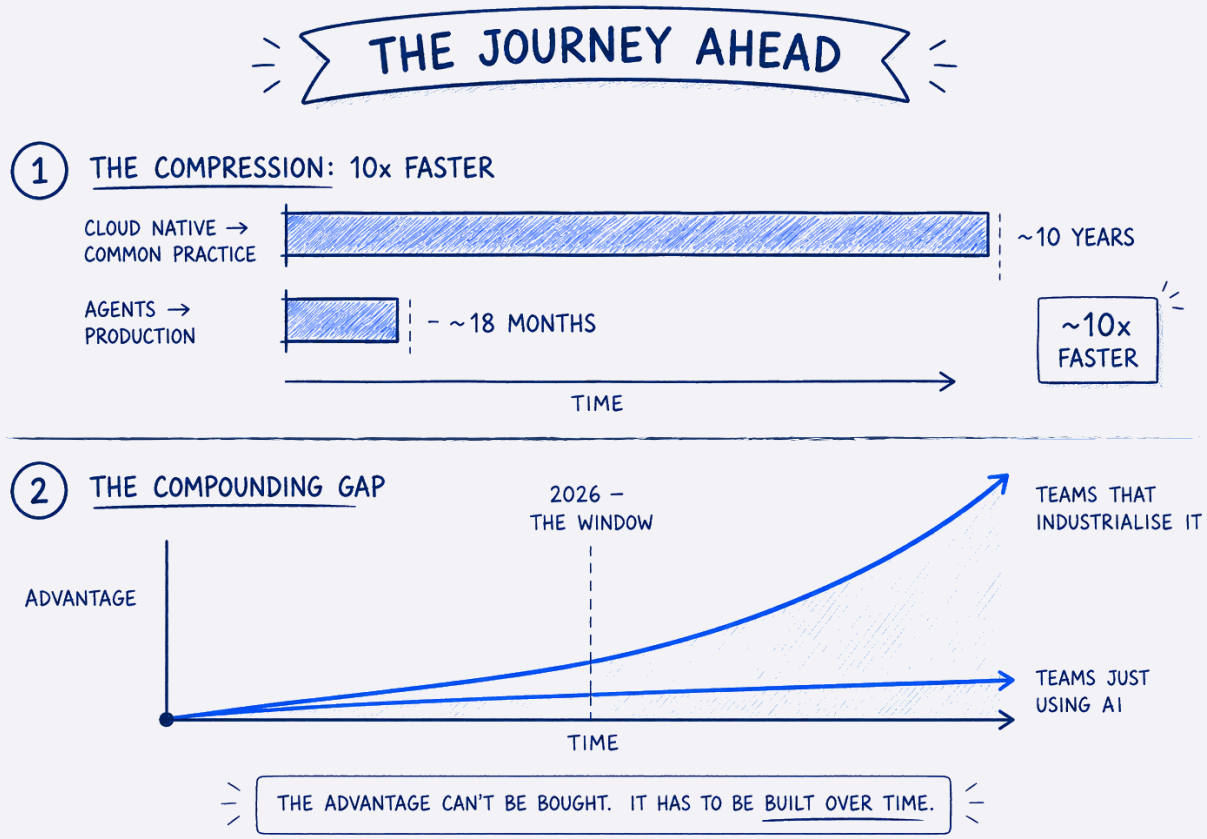
*The platform doesn't replace team autonomy – it amplifies it.*



# What Comes Next?

Cloud Native took roughly a decade to settle into common practice. The shift to containers and Kubernetes wasn't done overnight, and the teams that moved first ended up with operational advantages the rest spent years catching up to.

Agents went from research demos to production systems in eighteen months, about a tenth of the time Cloud Native took. The teams figuring this out in 2026 will set the operational standard for the rest of the decade.




This time the advantage compounds further. Cloud Native's advantages had a ceiling: once a team had Kubernetes, they could compete with another team running Kubernetes. The agent factory advantage compounds differently. The architecture, the context base, the validation harness, the team's understanding of what the agents can and can't do, none of it can be bought. It has to be built over time.

For engineering leaders reading this in 2026, the practical question is whether you're building one or competing with someone who is. The next twelve to eighteen months are when that question gets answered.

*The future won't be built by teams using AI. It will be built by teams that industrialise it.*

# Further Reading

There's more to read on this than this whitepaper covers. The primary sources for the implementations referenced throughout, plus a few pieces worth reading on adjacent topics: Steve Yegge's writing on "gas towns," the broader harness engineering literature, the spec-driven development tooling ecosystem, and the security guidance now emerging from standards bodies.

### The Implementations

Stripe Engineering, Minions Part 1:
https://stripe.dev/blog/minions-stripes-one-shot-end-to-end-coding-agents

Stripe Engineering, Minions Part 2:
https://stripe.dev/blog/minions-stripes-one-shot-end-to-end-coding-agents-part-2

Ryan Lopopolo at OpenAI, Harness Engineering:
https://openai.com/index/harness-engineering/

### The Engineering Practice

Martin Fowler, Harness Engineering:
https://martinfowler.com/articles/exploring-gen-ai/harness-engineering.html

Martin Fowler, Spec-Driven Development Tools:
https://martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html

### Spec-Driven Development Tooling

GitHub Spec-Kit:
https://github.com/github/spec-kit

Ihaig/intent:
https://github.com/Ihaig/intent

### Adjacent Patterns and Security Guidance

NSA Cybersecurity Information Sheet on MCP security (May 2026, U/OO/6030316-26):
https://media.defense.gov/2026/Jun/02/2003943289/-1/-1/0/CSI_MCP_SECURITY.PDF

Steve Yegge, Welcome to the Wasteland: A Thousand Gas Towns:
https://steve-yegge.medium.com/welcome-to-the-wasteland-a-thousand-gas-towns-a5eb9bc8dc1f

# Building one, or competing with someone who is?

We help enterprise engineering organisations design and build software factories – the architecture, governance, and validation behind agentic development at scale. Wherever you are on that path, we should talk.

REACH OUT > **hello@re-cinq.com**

re:cinq

re:cinq is an engineering consultancy helping medium and large organisations adopt AI across software development: building software factories, training teams, and modernising legacy systems.