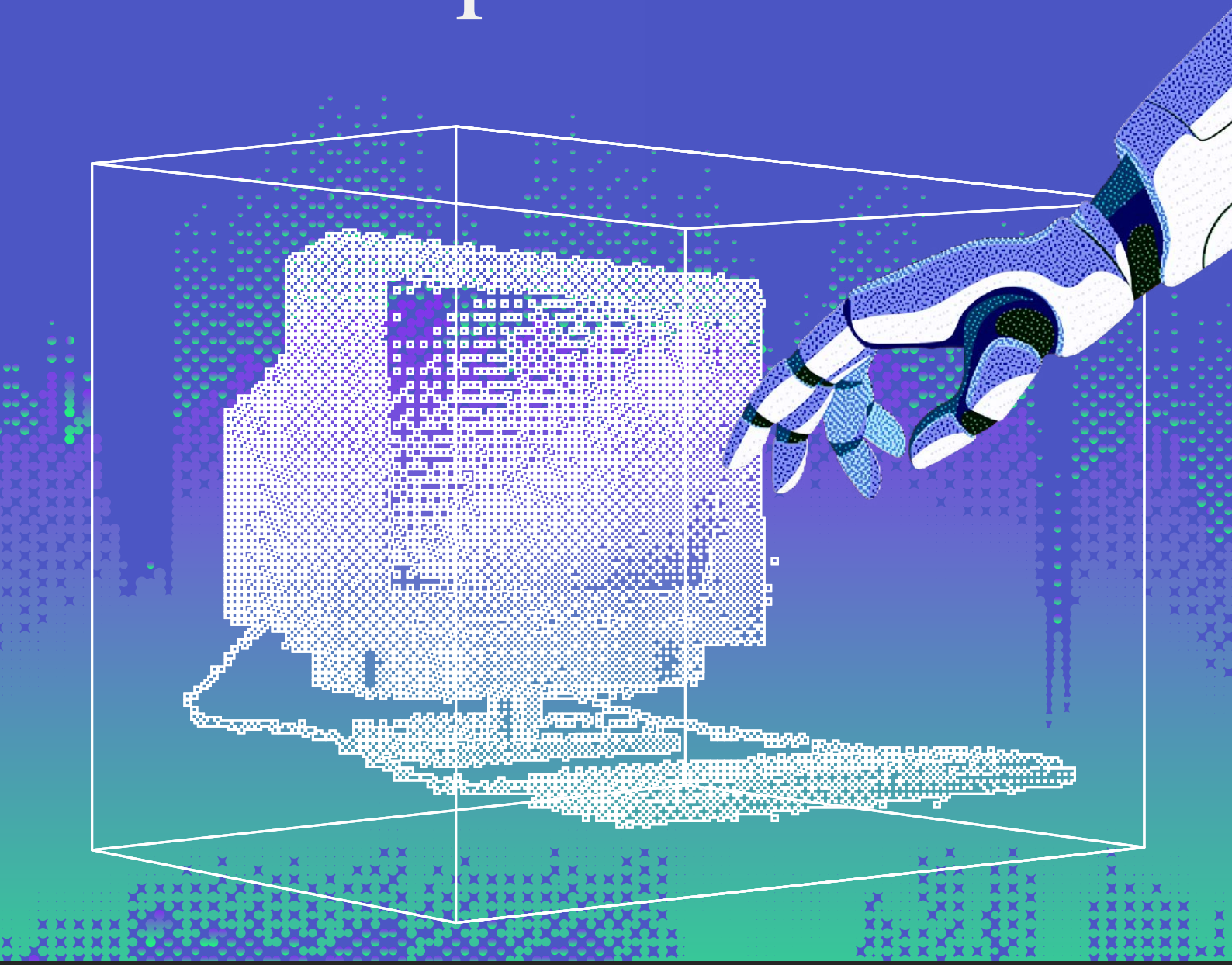
Weave Intelligence

The four levels of agentic software development in the enterprise



WEAVE INTELLIGENCE PRESENTS
THE FOUR LEVELS OF AGENTIC SOFTWARE DEVELOPMENT IN THE ENTERPRISE
PUBLISHED IN 2026

The four levels of agentic software development in the enterprise

Table of contents

Executive summary

Software engineering is entering a throughput race where output is no longer driven by human keystrokes. As the role shifts from writing code to overseeing agents, enterprise engineering organizations are encountering a new operational reality. The bottleneck isn't developer capacity anymore; it's the quality of the platform, and how safely and effectively it can work with AI.

This whitepaper explores that shift and what it means in practice. It outlines how organizations will need to rethink production systems design to stay competitive. Access to more powerful AI models is quickly becoming a commodity, with the real differentiator being how well an organization can manage and govern probabilistic intelligence at scale. To achieve this, the whitepaper details the necessary transition from traditional Internal Developer Platforms (IDPs) to Agentic Development Platforms (ADPs). An ADP is engineered to harness AI intelligence safely by deeply integrating probabilistic systems (foundation models and coding agents) with deterministic systems (CI/CD pipelines, policy enforcement, and ephemeral environments).

For engineering leaders navigating this shift, this document provides a roadmap and defines the four levels of agentic software development. These levels chart the maturity of an organization's platform, defined by the changing nature of human involvement across the feature development value stream:

THE FOUR LEVELS OF AGENTIC SOFTWARE DEVELOPMENT IN THE ENTERPRISE

LEVEL 0	Human is the loop	The baseline state. Human developers manually write code on a functional IDP with no agentic capabilities.
LEVEL 1	Human in the loop	Agents act as assistants inside the development environment via prompt-driven code generation and context retrieval. The human developer remains the primary execution engine.
LEVEL 2	Human on the loop	Agents become active, parallel participants in the value stream. A new path emerges to dispatch work to agents, which generate pull requests independently. Crucially, validation transforms from a human-driven gate into an industrial feedback loop, where agents execute repeatedly until the platform's deterministic checks pass.

LEVEL 3	Humans as orchestrators	The platform begins to operate as a continuous background execution layer. Agents respond to system signals (such as anomalies or dependency updates) without explicit human initiation. Human review becomes exception-based, and promotion decisions for low-risk changes become rule-based.
LEVEL 4	Fully autonomous	The production system becomes partially self-adjusting. Agents continuously monitor the environment and autonomously initiate, implement, validate, and promote changes in response to telemetry within predefined guardrails.

The era of human-in-the-loop software development is a temporary stepping stone. The organizations that succeed in the next decade will be those that evolve their deterministic platform layers and allow agents to work safely. This whitepaper provides the blueprint for building the industrial feedback loops needed to safely unleash agentic software development in the enterprise.

Introduction

Over the past few months, our research has revealed two distinct types of platform engineers. Those in denial over the rise of agents in software development, and those already deploying them and realizing breathtaking gains in productivity.

We're convinced that we are at the forefront of the biggest transformation the world of software engineering has seen. At a scale far greater than the rise of cloud computing. The traditional job of the software engineer has ceased to exist since Anthropic launched Opus 4.5 at the end of November 2025, and is now shifting from writing software to overseeing agents.

> **THE REAL BOTTLENECK**
>
> Access to powerful AI models is becoming a commodity. The real differentiator is no longer how fast we can write code, but how well an organization can manage and govern probabilistic intelligence at scale. The bottleneck has shifted from developer capacity to the quality and deterministic fabric of the platform.

We are entering a throughput race. Contrary to common belief, this isn't a race to "adopt the latest AI model", that's a given commodity. It's a race to multiply software output through parallelizing agents in software development, which is only possible if the platform allows them to act quickly and safely.

In this race, the traditional job description of the software developer as the person who manually codes, reviews diffs line by line, and incrementally advances features, is dissolving. Not because software disappears, but because human keystrokes are no longer the primary production mechanism.

In other words, the bottleneck is now the platform's quality and its interaction with AI models.

If your competitors can orchestrate ten agents in parallel while you still review every line manually, they're not ten percent faster. They move on a different curve, and that curve compounds. What changes is the role of coding. What doesn't change is the need for a platform production system. And the role and job profile of the platform engineer become drastically more important.

We acknowledge the existence of critical perspectives questioning the scalability of AI-first software development workflows within an enterprise context. Key concerns often revolve around security and control, the maintainability of an auto-generated codebase due to potential hallucinations, and the inherent risk of spiraling token costs.

Yet we currently see neither technical nor structural reasons why this shouldn't be solved, and in many organizations, agent-first development is already the predominant method. AI on its own is not the answer to those concerns. But when we analyzed failed AI initiatives in organizations, in the majority of cases, the cause wasn't the models. It was the provided guardrails and the platform's deterministic fabric that failed to provide checks and

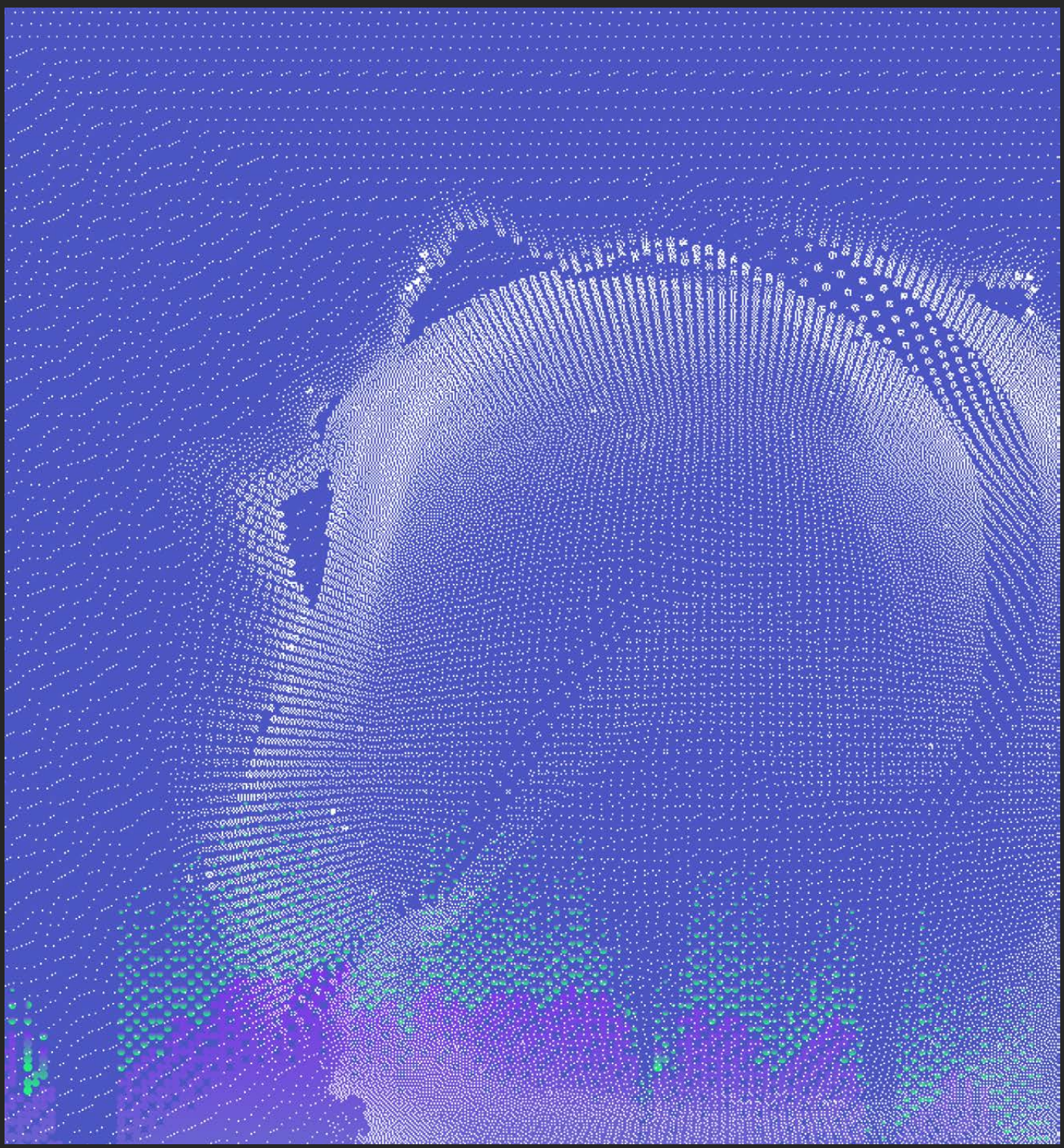
balances during execution.

> **WHY AI INITIATIVES FAIL**
>
> In a majority of failed AI initiatives, the cause wasn't the models themselves. It was the failure of the platform's deterministic fabric to provide the necessary guardrails, checks, and balances during execution.

Organizations that don't go all in on agentic development and don't radically change their underlying platform will be caught in a self-confirmation loop. They will simply layer AI on top and see all the downsides with little benefit. They then conclude "this isn't for me," reduce their effort, and ultimately fall significantly behind.

In this research, we propose a four-level framework for agentic development based on our observations from working with software engineering teams worldwide.. Those levels are differentiated by the degree of human involvement and range from level zero, with humans as the loop ("traditional" software development on a modern Internal Developer Platform (IDP)), to fully autonomous agents that ship software in response to environmental changes.

From assistance to orchestration



From assistance to orchestration

AI moves into software development in waves, marked by both the technical advancement of models and organizational change to optimally embed the technology into teams' workflows. This is usually characterized by a lag between technological advancement and organizational adoption.

It's worth noting that organizations are at very different stages of adoption. While some of you may just be dipping your toes into your first models, others are already working with highly parallelized agents tackling different tasks simultaneously.

Yet if you look more closely, the first experiment usually starts by augmenting the human software development flow with Large Language Models (LLMs) assisted autocomplete, inline assistance, faster search, and well-defined tasks. Some teams do see minor productivity enhancements, but they are negligible at scale.

The next level of applying AI is handing tasks off to agents. The developer describes the target state, and a specialized agent acts and delivers results back to the human. Once this works stably with a single agent, the next step is usually to coordinate teams of agents operating in parallel, reasoning across separate context windows, and executing multi-step workflows autonomously[1]. While agents do not always return accurate results, performance on SWE-bench Verified, the industry standard benchmark for autonomous software engineering tasks, has risen from near-zero in 2023 to above 60% for leading models as of early 2026, with no sign of deceleration.

Yet this, of course, highlights the need for policy-checking mechanisms to catch those errors and return them to the agent with a description of what to improve. These agentic improvement loops between several models or models and deterministic policy mechanisms will be the cornerstones of getting platforms agent-ready.

This transition from using AI as assistance to actually letting agents "do the work" creates friction. In early 2025, some teams saw a 19% slowdown due to AI noise. But by early 2026, experienced developers who shifted to a long-horizon agentic workflows reported 18% to 38% speedups[2].

> *88%*
>
> **of firms experiment with AI, only 5.5% translate that into a significant impact on Earnings Before Interest and Taxes (EBIT)**

At the macro level, the pattern is sharper, underscoring the urgency for enterprises to embed agentic AI into the very fabric of software development. McKinsey reports that while 88% of firms experiment with AI, only

5.5% translate that into a significant impact on Earnings Before Interest and Taxes (EBIT). High performers are five times more likely to make structural AI investments[3].

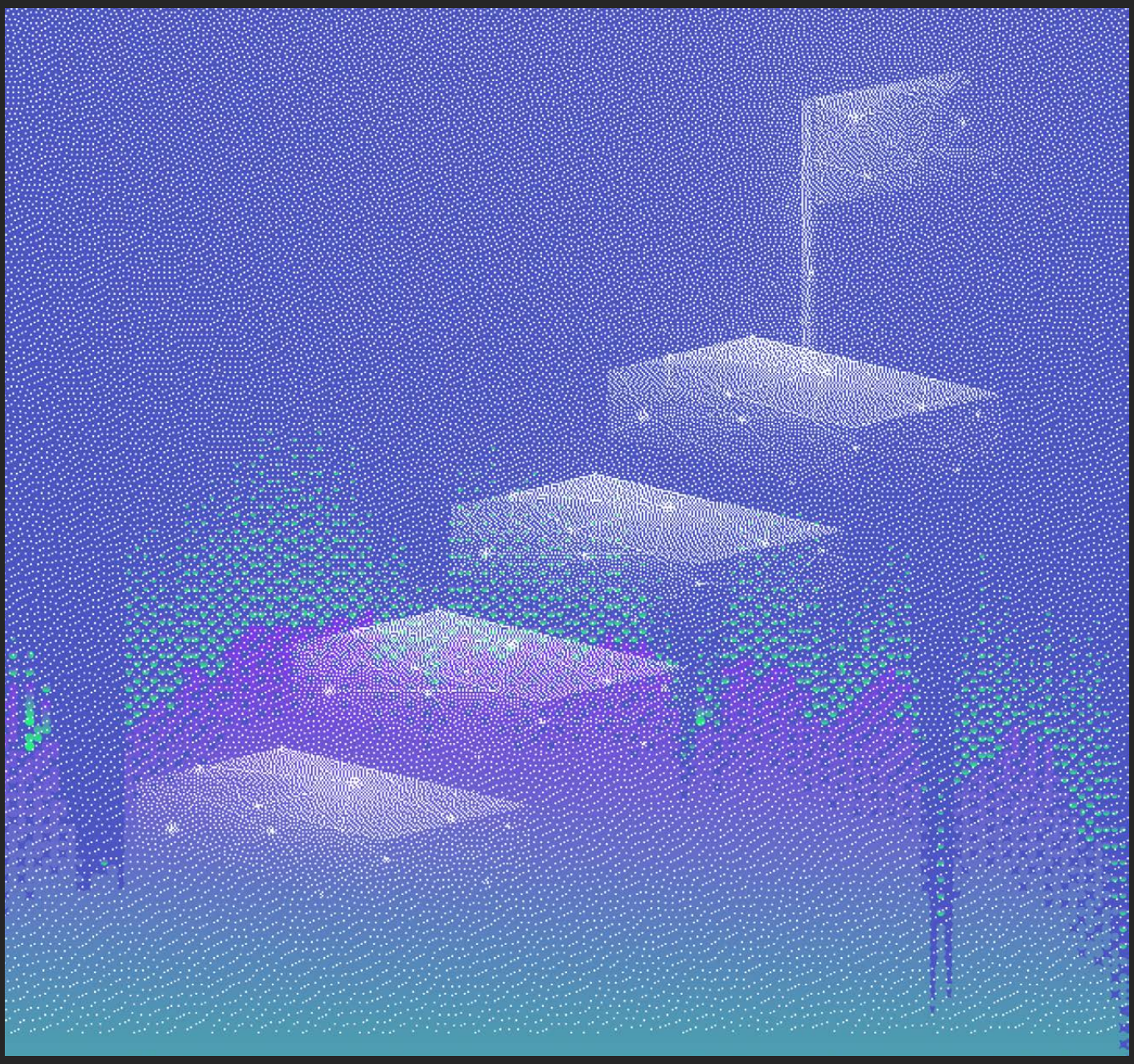
Many organizations understand that this race is happening at breakneck speed. Based on our conversations with enterprises, we predict that by the end of 2026, over 50% of enterprise software teams with a platform engineering team will have begun remodeling their platforms to support agent-first development.

> By the end of 2026, over 50% of enterprise platform teams will be architecting for an agent-first future.

While all of this is ongoing, we're aware of at least a handful of organizations already working on fully autonomous software development. Almost like high-frequency trading, the agents are watching customer interactions, feedback, and environmental changes, and proactively updating or adding features to the software.

From all the interviews and conversations we've had over the last two years, we identified four levels of agentic development that help software teams understand where they are today and what needs to change to reach the next level. We'll complement this research over the next few months with hands-on guides for transitioning from one level to the next from an architectural and change management perspective.

The four levels of agentic software development



The four levels of agentic software development

As organizations climb the ladder and mature in their agentic coding practices, the role of the human changes. When we propose the levels of agentic development, we differentiate them depending on the human involvement.

At level zero, the human is the loop. This base level is not counted among the four levels of agentic software development. Software development at this point is the interplay between a human writing code and a deterministically functioning platform that supports execution, deployment, and testing. Gains are linear; human development bandwidth is the constraint. If your organization doesn't yet have a functioning IDP, the most important first step is building the deterministic foundation that standardizes paths for delivery, deployment, and observation. Rather than a prerequisite to thinking about agentic development, this is the same work, seen through a different lens. The paths you build today for human developers are the same paths that agents will eventually execute.

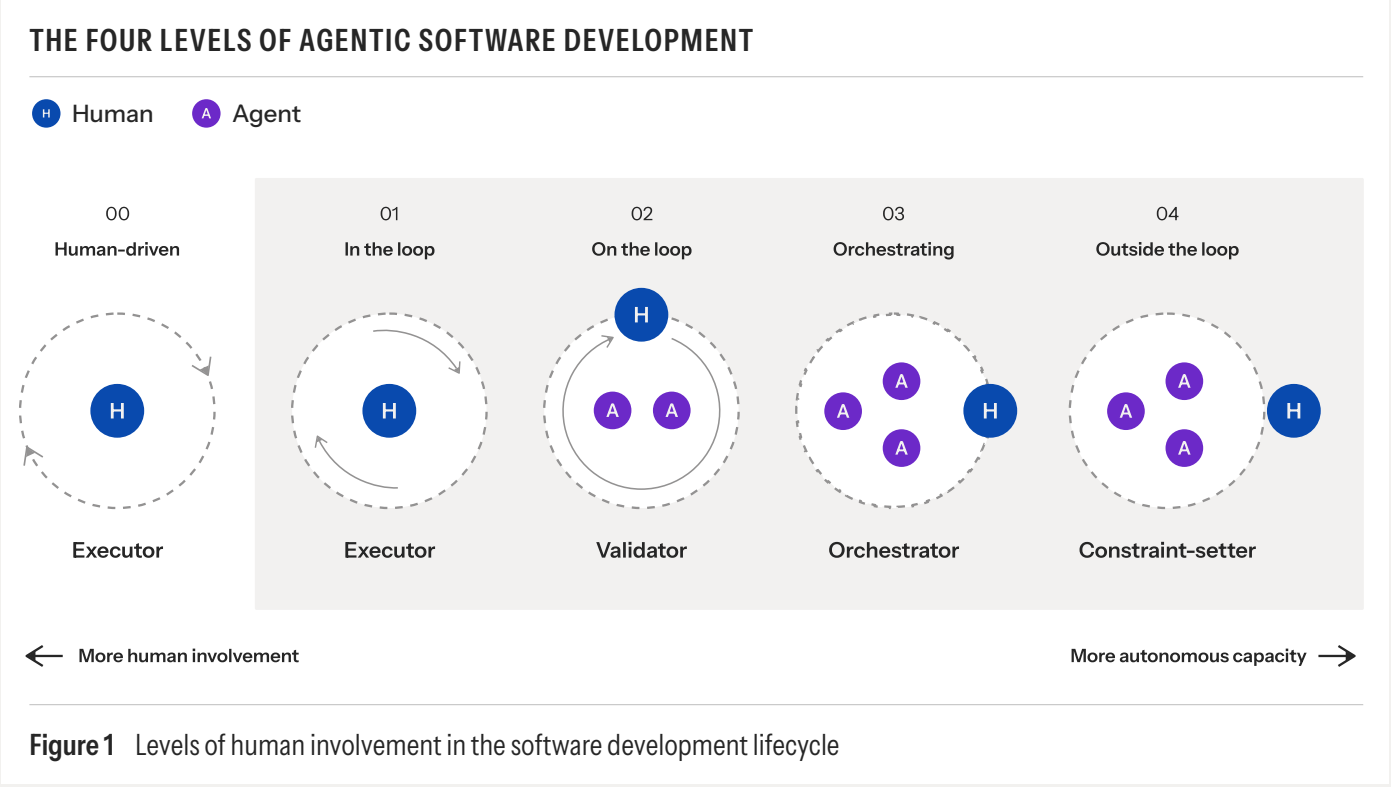


Figure 1 Levels of human involvement in the software development lifecycle

At the next (and first) level, the human is in the loop. Agents suggest, yet humans approve line by line. The developer remains the execution engine. Gains are still linear but higher, already in the 1.x gain region. It's still output restricted because human review bandwidth remains the constraint.

The most drastic change is likely the transition between the first and second levels. Humans increasingly move from being in the loop to only being on the loop. This is, of course, a continuum. At first, there are still many manual validation flows and code reviews before this is fully automated. But agents generate pull requests in parallel, and deterministic systems validate them. Humans still "trigger the change" and send agents off to do things. They also verify behavior rather than inspect every diff. Throughput increases because multiple paths advance simultaneously, so the bottleneck shifts from typing speed to validation quality.

At the third level, the human becomes the orchestrator of the loop. Agents execute long-running paths in the background, verification is largely automated, and human review becomes exception-based. The constraint shifts from review bandwidth to architectural maturity and token economics. The organization begins to operate a background execution layer. There is already a surprising number of organizations in between level two and three — or even fully on level three, at least for their frontend estate. This situation is far from a daydream.

And then there is a fourth level.

At this level, the system is fully autonomous for defined classes of work. Agents monitor external signals (customer feedback, telemetry, cost anomalies, security findings) and initiate new paths autonomously within predefined guardrails. They open tickets, propose changes, remediate issues, and adapt configurations without explicit human prompting. Humans define constraints, strategic direction, and escalation boundaries. Agents operate continuously inside that envelope. At this point, the traditional idea of software development collapses, and the agents are taking over the entire chain. What remains is the platform and with it the platform engineering team.

At this level, the production system becomes partially self-adjusting. The productivity curve bends upward at each step.

At this level, the production system becomes partially self-adjusting. The productivity curve bends upward at each step.

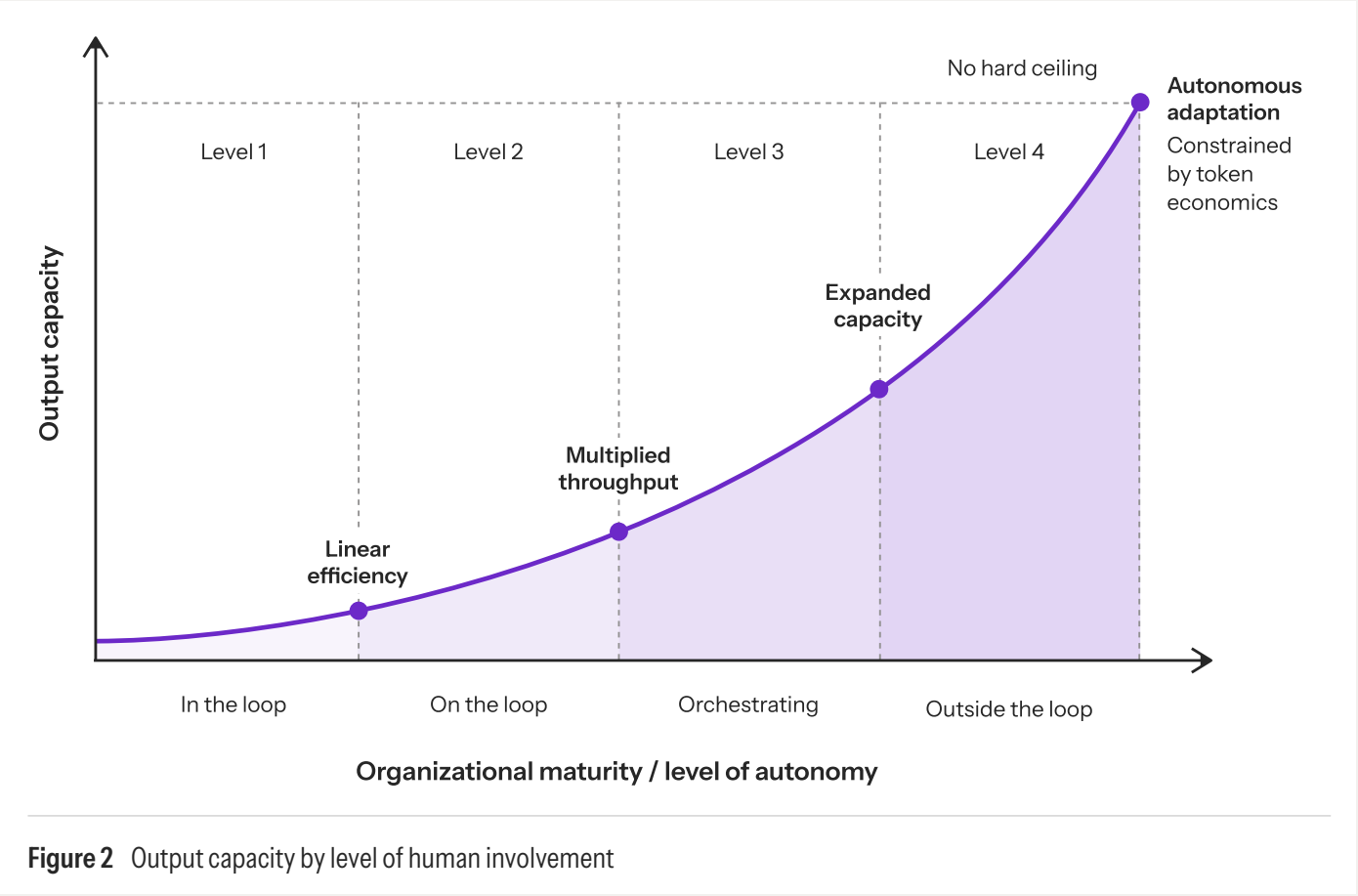


Figure 2 Output capacity by level of human involvement

Level one improves efficiency, level two multiplies throughput, level three increases execution capacity, and level four introduces autonomous adaptation.

The optimization constraint becomes clear: the lower you can safely reduce human intervention, the higher the output capacity. But safety depends entirely on the quality of the underlying platform. The impact on productivity increases by level. While we have anecdotal insights into how much this uptake can be, we lack reliable data.

No level is obtainable without an Agentic Development Platform (ADP)

It's tempting to think this is a model race, but it is not. Models are becoming a commodity and are already at a level similar to human coding capabilities with continuous incremental gains. Capability is converging. The differentiator is not coding intelligence. It is the production system that harnesses the intelligence and provides checks and balances to keep it safe and ensure predictable outputs.

Where today Internal Developer Platforms (IDP) help humans ship software predictably and at scale, agents require an advanced platform we call the Agentic Development Platform (ADP).

An IDP standardizes paths for humans following platform-as-a-product principles; an ADP makes those paths executable by agents at scale and eventually allows agents to initiate them autonomously. To enable this, the architecture must integrate two domains that operate under different logics.

On one side sit the probabilistic systems:

- → **Foundation models**
- → **Coding agents**
- → **Multi-agent coordination frameworks**
- → **Model-evaluating patterns**
- → **Behavioral evaluation layers**

These components reason, generalize, and improve with context. And while they are very powerful, they are not predictable.

On the other side sit the deterministic systems:

- → **Platform orchestration that ensures safe creation and update of infrastructure components**
- → **Explicit capability boundaries and stable APIs**
- → **CI/CD pipelines with reproducible execution**
- → **Ephemeral environments for isolation and verification**
- → **Policy enforcement mechanisms**
- → **Identity and Role Based Access Control (RBAC) management**
- → **State management**
- → **Ontology or semantic access layers over proprietary data**
- → **Dependency and license verification**
- → **Observability and audit trails**

How advanced an ADP needs to be depends on the level of agentic development you're aiming for. The architectural design will go through a transition.

We've published iterations on the reference architecture for IDPs over the last few years, and we'll soon publish a reference architecture for an ADP[4]. Yet more telling than the platform's architecture is the paths that software developers and later agents use to interact with the platform to achieve a specific outcome within a defined value stream. We'll use the paths to outcomes model in this case.

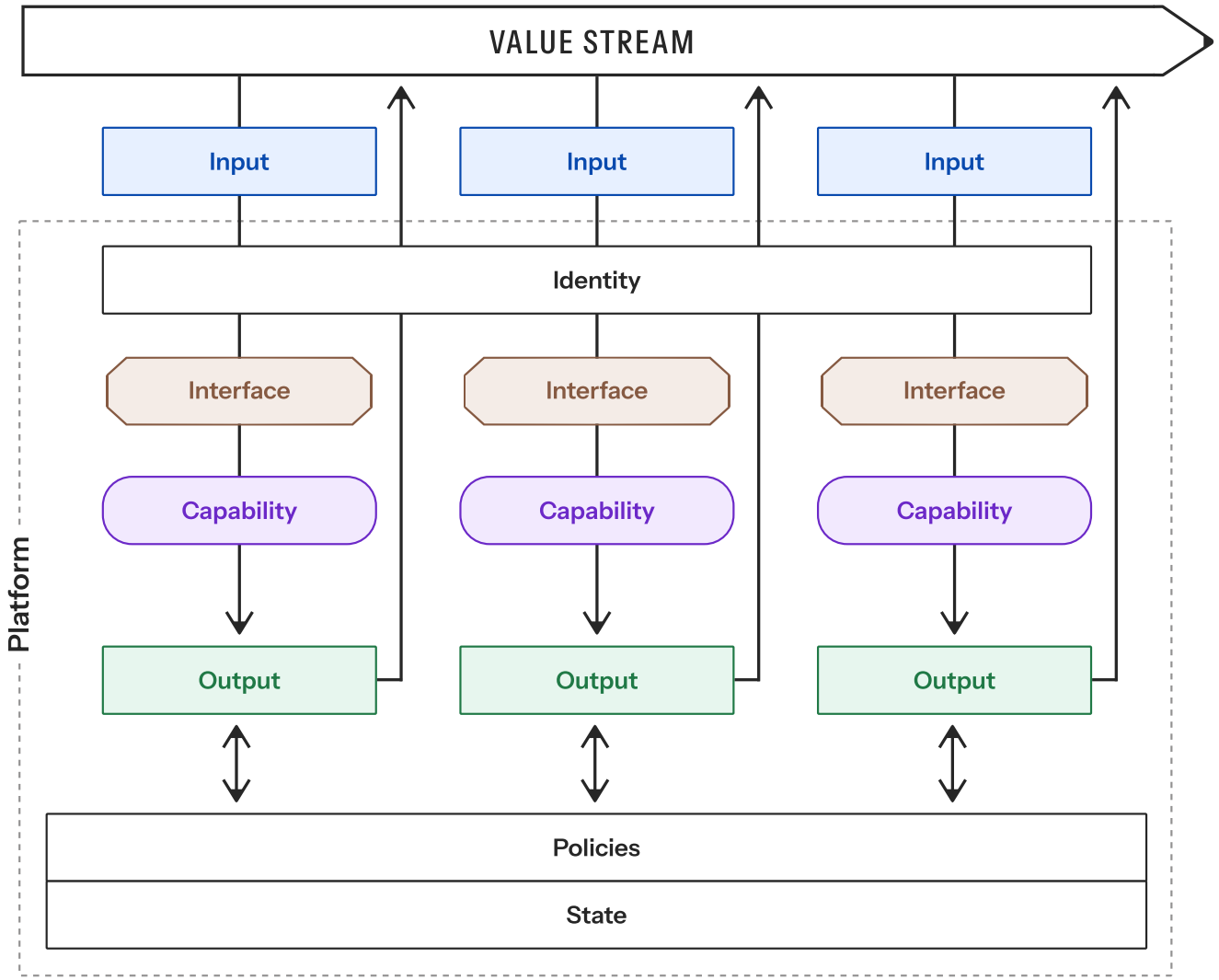


Figure 3 Paths to outcomes model

We'll next walk through the different levels and discuss what paths a well-rounded platform should expose alongside the value stream of "delivering a feature". The idea of paths alongside a value stream that humans or agents can use as needed fits better.

Level 0: Paths in software development today

Level 0 characterizes the base case for software development, where no agents are involved but a functional IDP is already in place. In this case, we assume the most common paths on the value stream “feature development”.

The table below summarizes the most common paths in the feature development value stream at Level 0.

PATH	TYPICAL INPUT	EXAMPLE CAPABILITIES INSIDE THE PATH	OUTPUT
Retrieve context	Ticket, issue, or feature request	Repository search, documentation retrieval, dependency lookup	Context for the developer
Implement change	Code creation	Editing tools, compilers, dependency managers	Candidate code change or commit
Validate change	Code change or pull request	CI pipelines, unit tests, integration tests, security scans	Verified artifact or failing validation
Promote change	Validated artifact	Review systems, promotion policies, environment checks	Approved artifact ready for deployment
Deploy system	Release artifact	Deployment orchestrators, infrastructure provisioning, rollout automation	Running system in staging or production
Observe system	Runtime signals	Logging, metrics, tracing, alerting	Operational visibility and telemetry
Remediate issue	Alert or operational signal	Debugging tools, rollback mechanisms, patch deployment	Restored system state

In our standard diagram, it will look as follows:

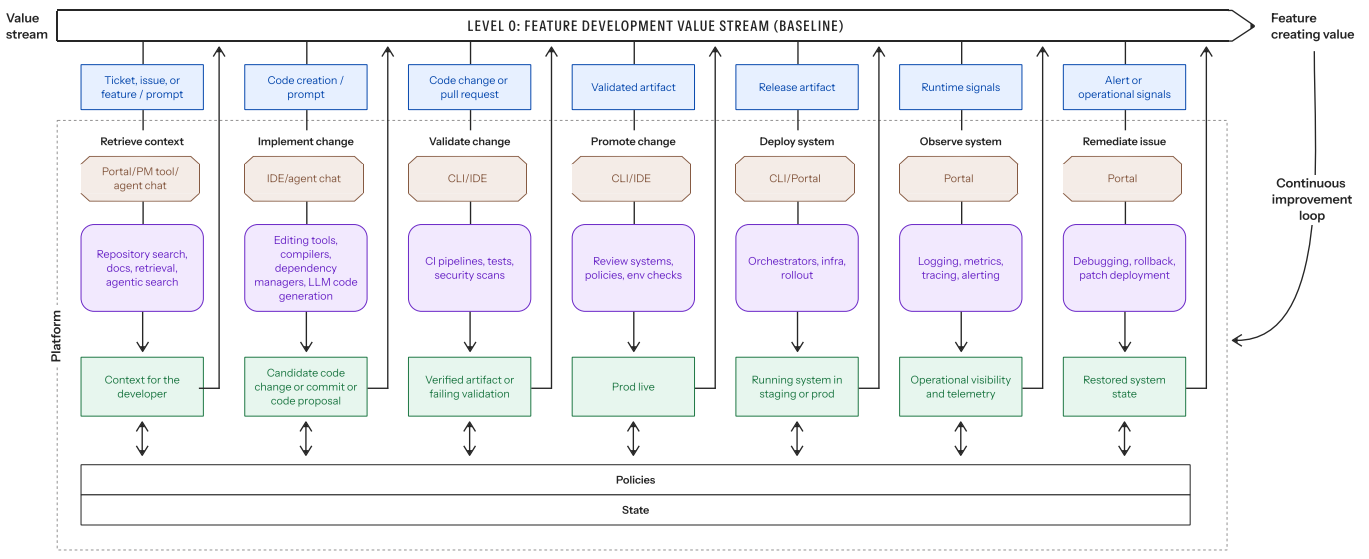


Figure 4 Level 0: Feature development value stream (baseline)

The baseline IDP, as displayed, uses the paths to outcomes model.

Seven paths transform value stream inputs into outputs:

- → **Retrieve context**
- → **Implement change**
- → **Validate change**
- → **Promote change**
- → **Deploy system**
- → **Observe system**
- → **Remediate issue**

This stage is the starting point before any agentic capabilities are introduced. These paths form the operational backbone of modern IDPs. Each path encapsulates a recurring intent: obtain context, implement a change, verify correctness, promote the change, deploy the system, observe behavior, and remediate issues.

The tools used to implement these paths may vary across organizations and technologies. A CI pipeline might beJenkins, GitHub Actions, or GitLab. Deployment might happen through Kubernetes, serverless infrastructure, or virtual machines. Yet the structure of the paths remains the same. This stability is precisely what makes the model useful. It allows us to reason about platforms independently of specific tools or architectures.

Level 1: Human in the loop

The first level of agent adoption does not fundamentally change how software is produced.

The production system remains human-driven. Engineers still design systems, own the codebase, review changes, and decide what enters production. The change is how certain parts of the work within a path are executed, most notably during implementation and system exploration.

Agents begin assisting developers within the development environment. Instead of writing every line of code manually, engineers increasingly delegate parts of the implementation to coding agents that generate, modify, or explain code based on prompts.

The important observation is that the surrounding production system remains intact.

The same repositories, CI pipelines, deployment systems, and security controls continue to govern how software moves toward production. Agents operate inside that system, not outside it. Seen through the path to outcome model, this means that the paths in the feature development value stream remain unchanged.

A developer still moves from a requirement to an implementation, from implementation to validation, and from validation to deployment. The change is that some capabilities within these paths now include probabilistic systems.

In particular, the implement change path evolves. Where developers previously wrote code manually, they now increasingly collaborate with coding agents that generate and modify code based on prompts.

Changes in the feature development value stream (Level 1)

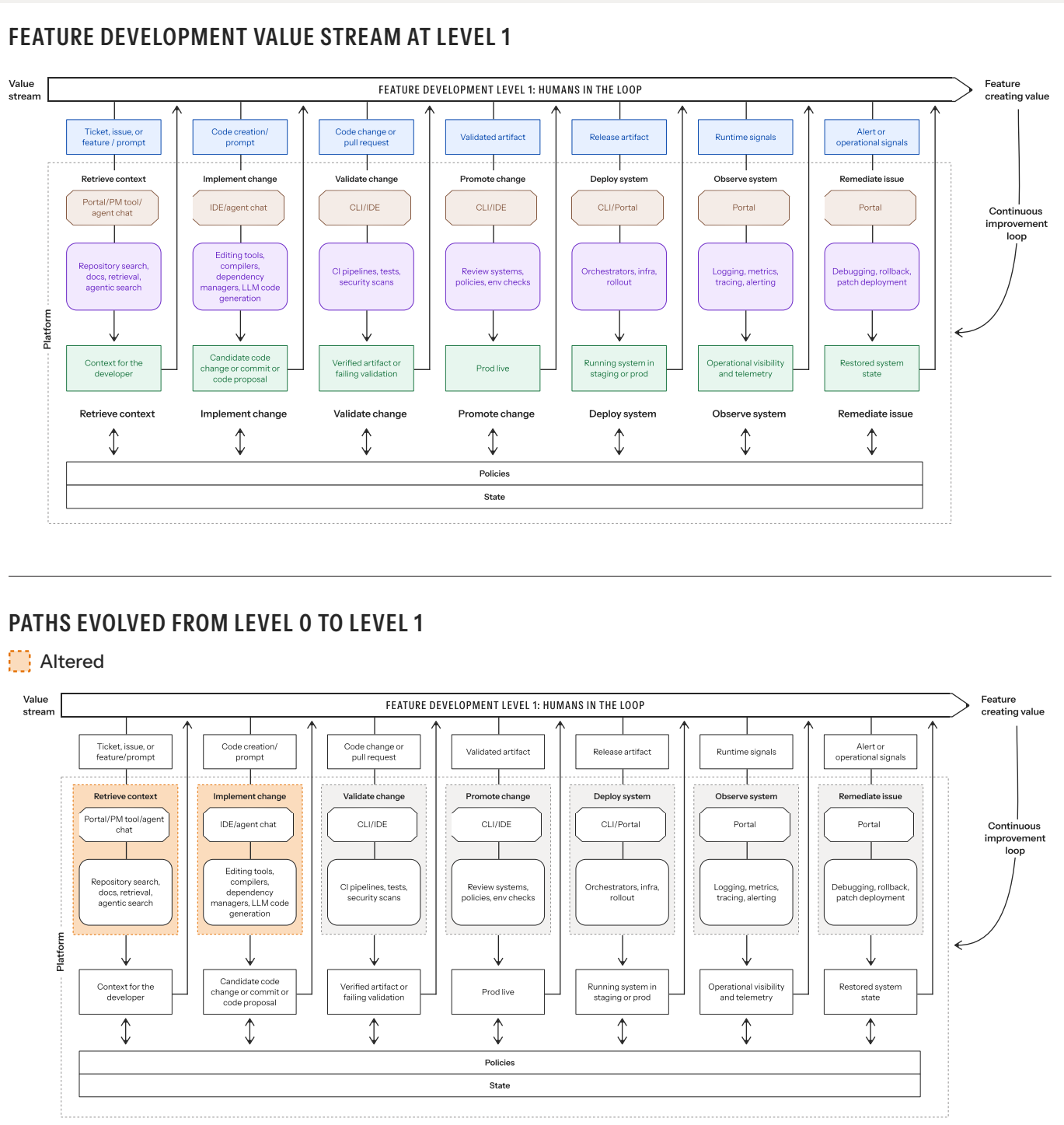


Figure 5 Level 1: Humans in the loop with two paths altered. Retrieve context adds agentic search, and implement change adds LLM code generation.

From the perspective of capabilities, no new paths are added, and no paths are removed. We have only two changes to the implementation path. These changed capabilities assist the developer but don't yet control the path.

The most visible capabilities introduced at this level are autocomplete and prompt-driven code generation. A developer describes the intended change through a prompt. The agent generates code that implements part of the feature. The developer inspects the result, modifies it where necessary, and integrates it into the codebase. The important aspect is that the developer remains responsible for accepting or rejecting the generated output.

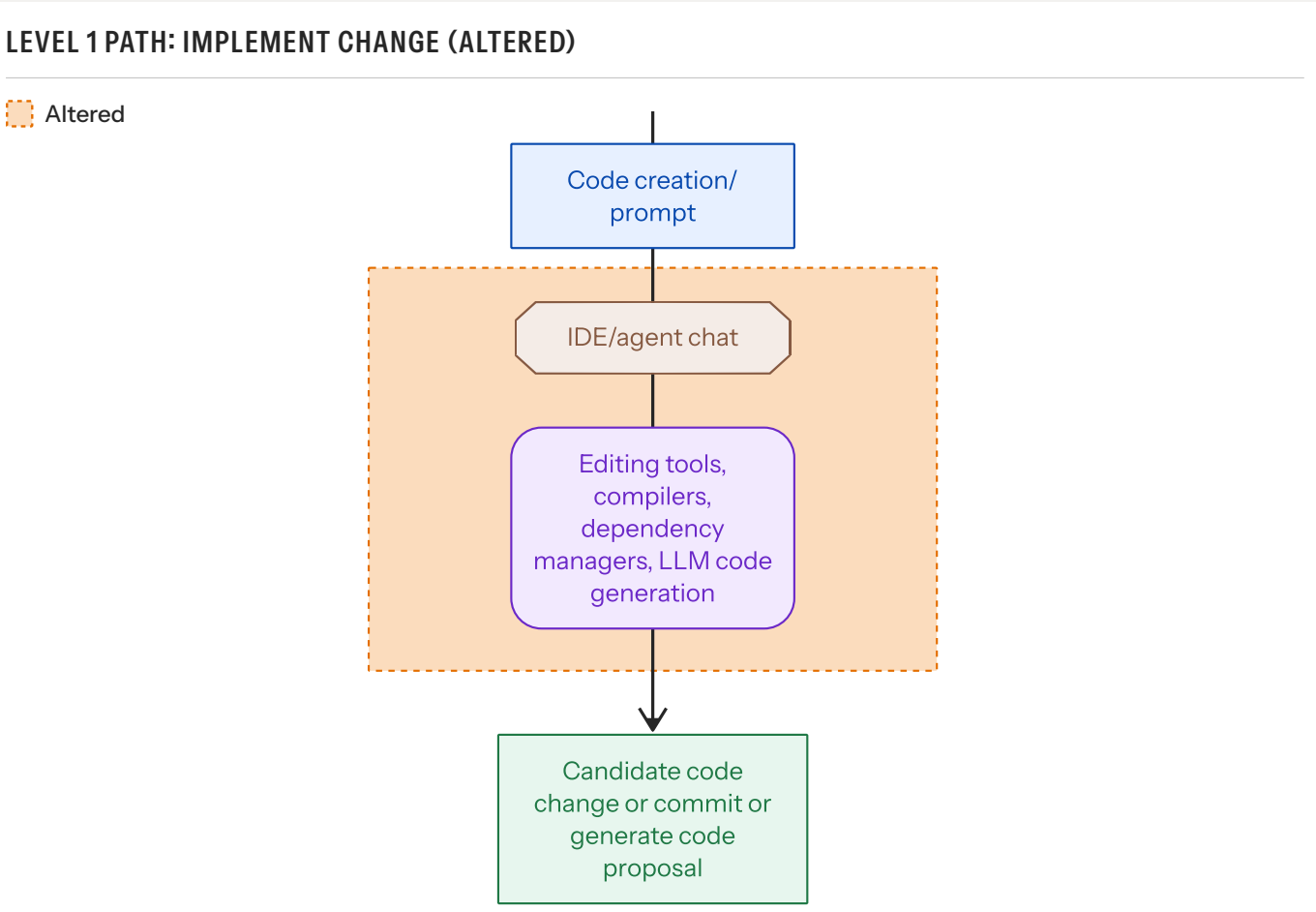


Figure 6 Level 1: Implement change path - The Integrated Development Environment (IDE) gains an agent chat interface alongside traditional editing tools. LLM code generation joins compilers and dependency managers as a capability. Output expands from just candidate code changes to include generated code proposals that developers review and refine.

A second capability appears on the retrieve context path when developers use agents to understand unfamiliar parts of the system. Instead of manually searching through documentation and source code, the developer prompts the agent to explain how a component works, summarize relevant files, or propose a modification strategy. The agent produces a structured explanation that helps the developer continue their work.

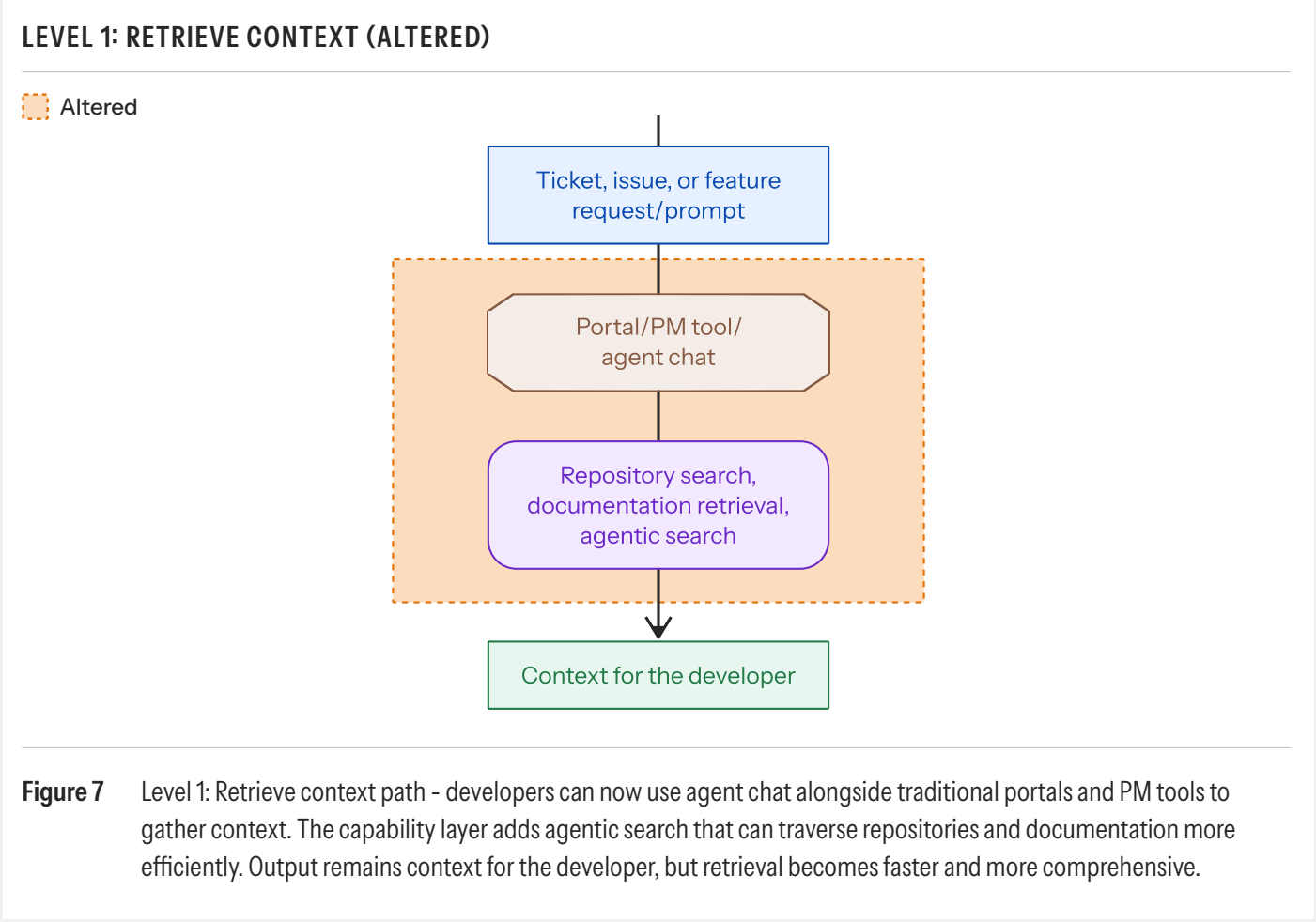


Figure 7 Level 1: Retrieve context path - developers can now use agent chat alongside traditional portals and PM tools to gather context. The capability layer adds agentic search that can traverse repositories and documentation more efficiently. Output remains context for the developer, but retrieval becomes faster and more comprehensive.

Despite these new capabilities, the production system's structure remains unchanged. Code still moves through pull requests. CI pipelines still validate changes. Deployments still occur through controlled delivery systems. Observability systems still monitor production. The deterministic platform infrastructure remains the final authority.

Agents assist developers in producing changes, but they don't yet control how those changes move through the system. Level 1 represents a local augmentation of capabilities, not a structural transformation of the platform in the path to outcome model. It does, however, shield against the risk of hallucination and is significantly easier to operate. Of course, the productivity gains are limited.

The paths that govern software delivery remain the same. The change is that some of the capabilities within those paths now involve probabilistic systems that help humans perform their work more efficiently.

Level 2: Human on the loop

Level 1 improved developers' experience in their editor. Level 2 is a much more fundamental change that requires modifications across the platform.

Agents stop being a private tool on a workstation and become a participant in the value stream. They do whole batches of work, yet they're still being "sent off" by a human to do that. In other words, they aren't yet reacting to triggers; the trigger is the human, and the human remains accountable.

The simplest way to describe the shift is:

AT LEVEL 1

Work entered the system when a developer chose to type.

AT LEVEL 2

Humans are now sending off agents to do chunks of work on their behalf.

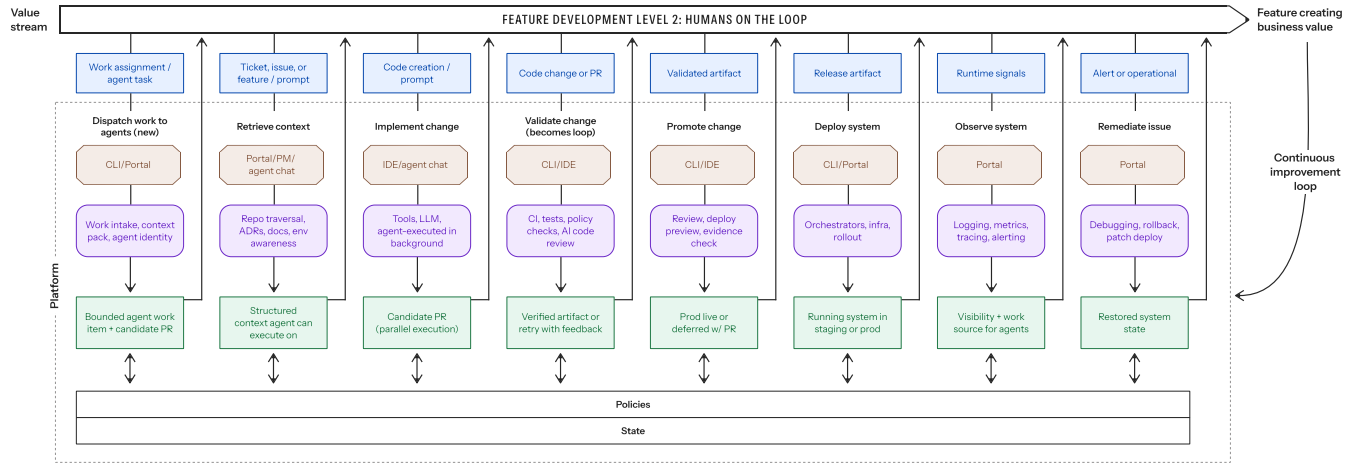
This scenario is the first real form of parallelization. Not multiple developers, but multiple work items moving forward simultaneously because the human can ask several agents to make progress in the background.

Changes in the feature development value stream (Level 2)

At Level 2, we still have the same backbone paths described at Level 0 and Level 1:

- → **Retrieve context**
- → **Implement change**
- → **Validate change**
- → **Promote change**
- → **Deploy system**
- → **Observe system**
- → **Remediate issue**

FEATURE DEVELOPMENT VALUE STREAM AT LEVEL 2



PATHS EVOLVED FROM LEVEL 1 TO LEVEL 2

Altered Expanded Unchanged

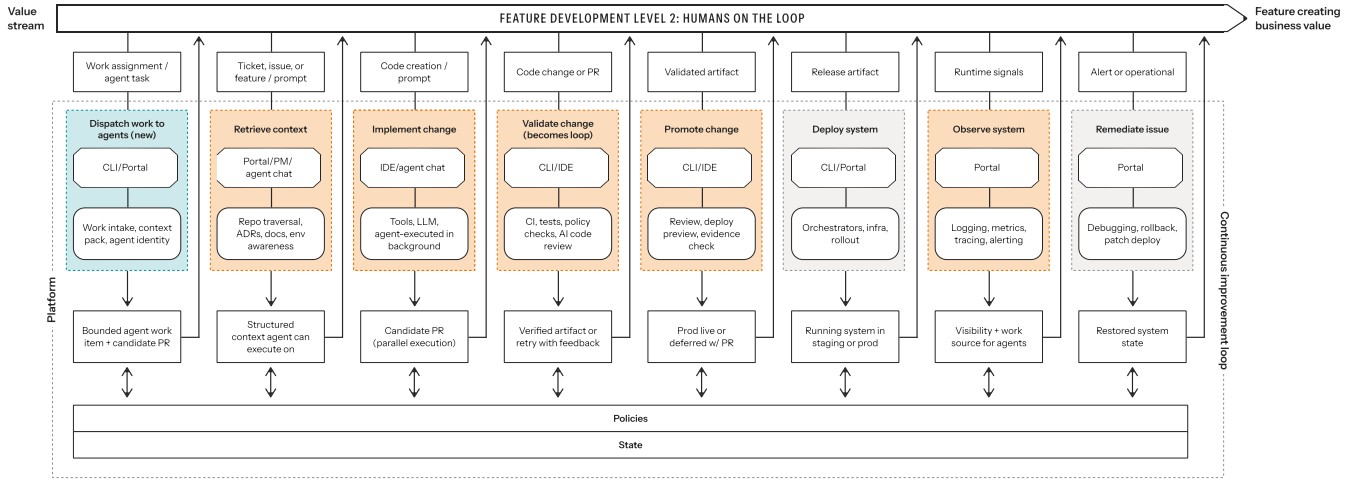


Figure 8 Level 2: Humans on the loop. The most significant structural shift is illustrated using the path to outcome model with eight paths. A new path (dispatch work to agents) enables parallel execution, while the validate change path becomes a loop where agents retry until passing. Six of eight paths are new or altered at this level.

No core path disappears. The production system is not replaced. What changes is:

1. **A new path appears that turns human-directed work assignments into parallel agent execution in a governed way.**
2. **Several existing paths change because agents become callers, and because validation becomes a loop rather than a single gate.**

Dispatch work to agents path (new at Level 2)

Let's first analyse the new dispatch work to agents path added at Level 2. This path takes human work assignments and turns them into agent-executable work items with the appropriate context, identity, and boundaries. This is the structural change Level 1 does not have. And this is also where parallel execution becomes operational.

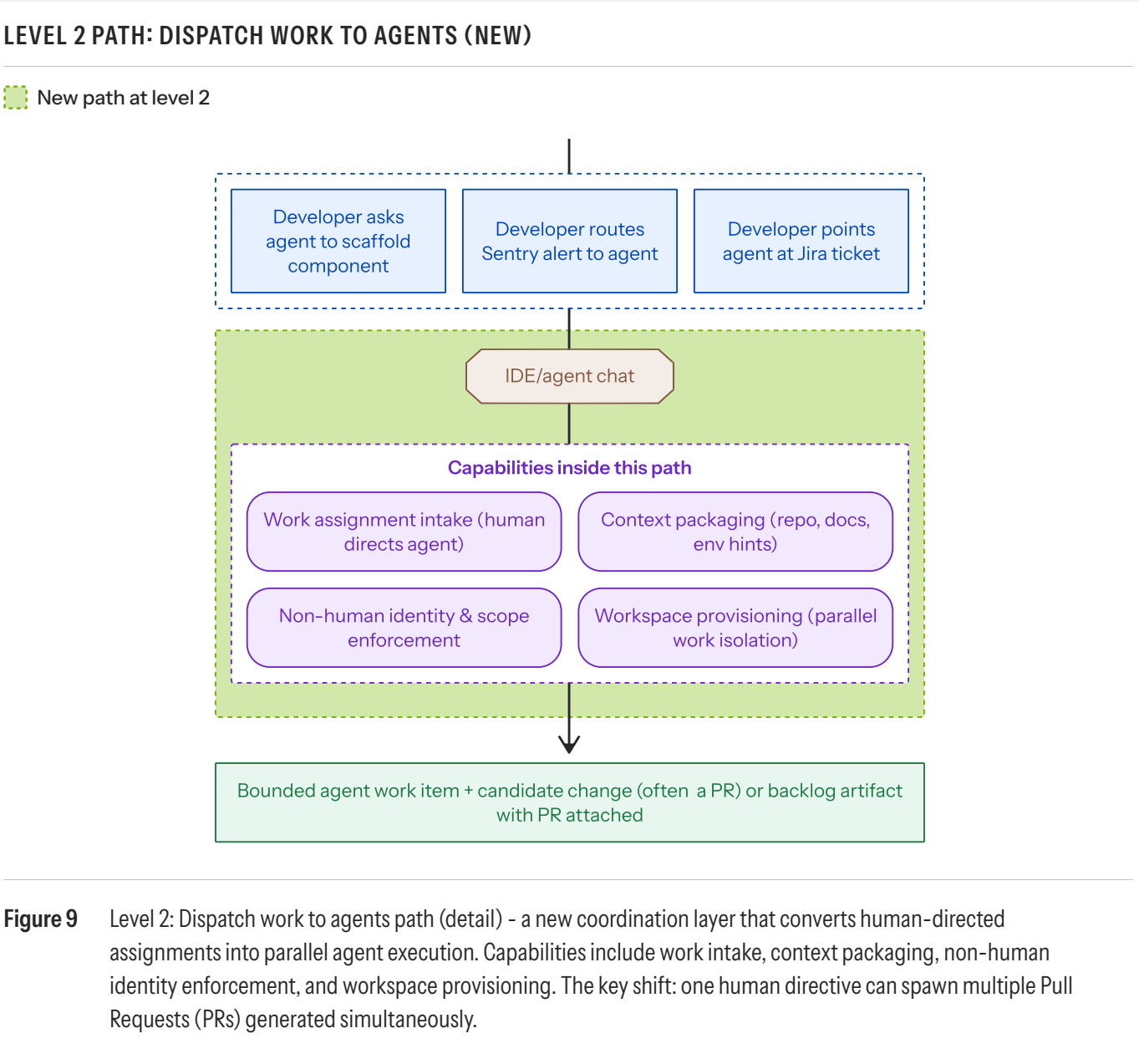


Figure 9 Level 2: Dispatch work to agents path (detail) - a new coordination layer that converts human-directed assignments into parallel agent execution. Capabilities include work intake, context packaging, non-human identity enforcement, and workspace provisioning. The key shift: one human directive can spawn multiple Pull Requests (PRs) generated simultaneously.

Typical inputs are not “requirements” in the Software Delivery Life Cycle (SDLC) sense. They are human-directed assignments:

- → **A developer asking an agent to scaffold a component based on a screenshot from a customer call .**
- → **A developer tagging an agent to investigate a Slack thread.**
- → **A developer routing a Sentry alert into an agent workspace.**
- → **A developer asking an agent to do pre-work on a backlog item before the sprint begins.**
- → **A developer pointing an agent at a Jira ticket to generate a first-pass implementation.**

The defining pattern: A human sends one or more agents to do work in an agent execution layer (e.g., the web interface of Claude Code in their case), where multiple PRs can be generated in parallel. The human decides later what to advance.

Capabilities inside this path:

- → **Work assignment intake (human directs agent to a task via UI, CLI, or integration)**
- → **Context packaging (“context stack”: repo, docs, known issues, environment hints)**
- → **Non-human identity and scope enforcement (what the agent may touch)**
- → **Workspace provisioning for agent runs (so parallel work does not collide)**
- → **Output routing (create a PR or create a ticket with PR attached)**

Output: A bounded agent work item plus a candidate change (often a PR) or a backlog artifact with the PR attached.

Retrieve context path (altered)

At Level 2, the retrieve context path still exists, but the primary consumer is no longer only the human. At Level 2, the agent must retrieve context reliably enough to act without constant human clarification. This is where documentation quality stops being a nice-to-have.

Capabilities inside this path now need to support agent-scale retrieval:

- → **Repository and dependency traversal that doesn’t explode context windows**
- → **Access to architectural intent (docs, Architectural Decision Records (ADRs), diagrams)**
- → **Environment awareness (what services exist, where they run, what policies apply)**

Output: Structured context the agent can actually execute on, not just prose.

Implement change path (altered)

At Level 1, the implementation change path was still human-owned with AI assistance. At Level 2, it becomes agent-executed for many work items. The human has sent the agent to do this work. The agent produces the candidate artifact: PR, patch, or configuration change.

There are two perspectives vital to this path. It scales because work happens in the background and in parallel, not locally. Multi-repo work exists but is still awkward: many teams handle it by having one agent propose changes and generate instructions for another repo.

Output: candidate change artifact, almost always a PR.

Validate change path (altered, becomes a loop)

This is the most underestimated shift in agentic development, so we need to spend a little more time on this. At Level 1, validation is still a gate that humans walk through. A developer submits a pull request, CI runs a set of tests, and reviewers inspect the change before allowing it to progress. The validation path remains the same. What expands dramatically is the number and depth of capabilities inside that path.

At Level 2 this structure breaks down.

Once agents begin producing pull requests in parallel (at the direction of their human operators), validation cannot remain a single checkpoint. The production system would immediately stall. Instead, validation becomes a loop that agents execute repeatedly until a change satisfies the platform's requirements.

When validation fails, the failure signals are routed back to the agent that produced the change. The agent then modifies the implementation and attempts again. This cycle may repeat multiple times before the system produces a candidate artifact that satisfies the verification mechanisms.

The aspiration is that the verification phase should eventually become boring. In other words, by the time a change reaches validation, it should almost always pass. If it doesn't, the system should treat that failure as feedback for improving the agent's operating model. The immediate response is to reprompt the agent to fix the issue, while the longer-term response is to update the agent's context stack, rules, or testing expectations so the same class of failure is prevented earlier in the process.

Validation transforms from a passive safety net into an industrial feedback mechanism. Instead of merely detecting problems, the system continuously pushes defect resolution upstream.

Three structural consequences follow.

The first is CI capacity pressure

Agents iterate far more aggressively than humans. They retry until validation succeeds. They also execute broader verification suites than developers typically run locally. This results in more pipeline executions, more ephemeral environments, and potentially dramatically higher compute consumption. If not carefully designed, CI infrastructure and cloud costs can scale faster than the productivity gains agents deliver.

The second is that validation must widen beyond traditional testing

Unit tests alone cannot replace the judgment previously exercised by human reviewers. Organizations, therefore, expand deterministic verification to include scenario execution, performance checks, dependency and license analysis, and environment integrity checks.

The third is security and policy enforcement

When agents produce changes at volume, manual security review does not scale. Security scanning, vulnerability detection, secret exposure checks, and compliance validation must be automated as part of the validation loop. The same applies to infrastructure configuration.

Policy checks on Infrastructure as Code (IaC) artifacts (Terraform plans, Kubernetes manifests, Helm charts) must run as deterministic gates, not as post-deployment audits. Tools like Open Policy Agent (OPA), Kyverno, or Checkov become part of the CI pipeline rather than optional additions. If agents can propose infrastructure changes, the platform must verify them against organizational policy before they proceed. While not new in principle, at Level 2 this becomes non-negotiable because the volume and speed of agent-generated changes make manual policy review impossible.

Well-defined, codified policies will also allow organizations to safely move from Level 2 to Level 3, because they make the rules machine-readable and machine-enforceable. The quality of the policy layer directly determines how much agent autonomy you can safely grant.

Another ordering change also appears.

AI-assisted code review is most effective when it runs early in the validation loop rather than at the end. Tools such as automated review agents can generate comments and improvement suggestions before a human ever looks at the change. The agent can then automatically incorporate those corrections. By the time the pull request reaches a human reviewer, most mechanical issues have already been resolved.

Human review, therefore, shifts in character. Instead of inspecting every line of code, humans verify the resulting system behavior and ensure the change aligns with architectural intent.

An important nuance emerges when comparing different types of systems.

Frontend development often relies on deploy previews running in ephemeral environments. In these cases, validation focuses on behavioral verification. Agents generate the change, preview environments render the result, and reviewers inspect the running feature rather than the code itself. Some teams even introduce usability agents that interact with the application in a browser and attach screenshots or videos demonstrating the feature in action.

Backend systems require a different emphasis. Because backend services enforce data integrity, security boundaries, and operational guarantees, validation must rely more heavily on deterministic checks. Automated end-to-end scenarios, performance tests, startup time regressions, and security policies become essential evidence before a change can progress.

Both approaches share the same principle: validation must produce evidence strong enough to replace manual diff inspection.

The following diagram illustrates how this validation loop operates once agents participate in the development flow.

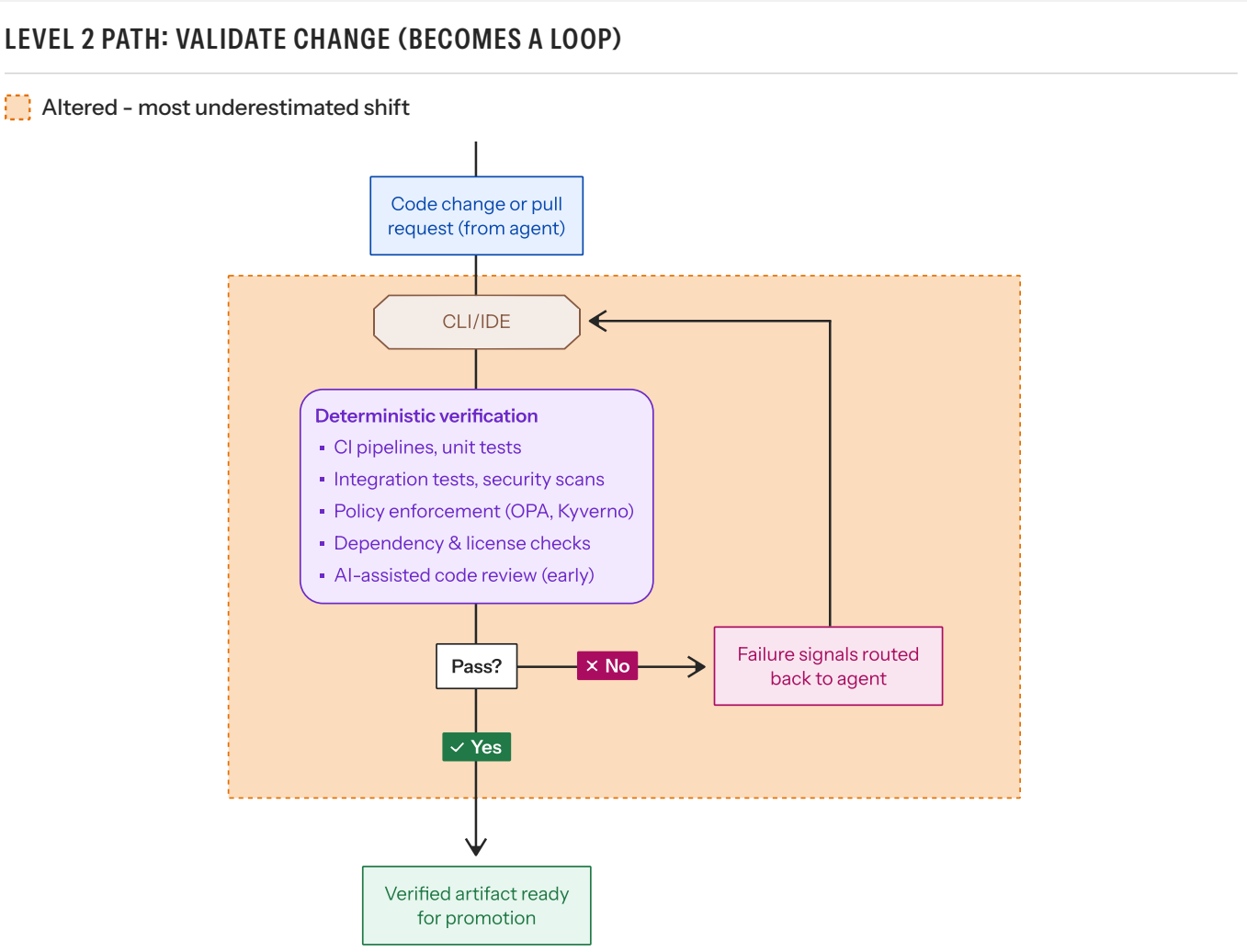


Figure 10 Level 2: Validate change as a loop - validation transforms from a gate humans walk through to a loop agents execute repeatedly. When validation fails, signals route back to the agent, which modifies and retries. The aspiration: by the time changes reach validation, they should almost always pass, and failures become feedback to improve the agent's operating model.

Promote change path (altered)

Promotion changes in what the human is doing. Humans are no longer reading every diff. They are verifying behavior, reviewing evidence, and prioritizing what to merge and ship.

Two prioritization gates that matter here: "Could we, in theory, ship this today?" "If it doesn't work, is it easy to fix?" If yes, loop back and improve the agent's rules. If no, backlog with PR attached.

At Level 2, promotion becomes the place where:

- → **Behavior is verified via deploy previews (especially on the frontend)**
- → **Evidence is reviewed (tests, scenarios, policy outcomes)**
- → **Work is triaged into "hip now vs park with PR attached.**

Output: Approved change ready for deployment, or deferred change packaged correctly.

Observe system path (altered)

The path exists at Level 0 and Level 1, but at Level 2, it becomes important as a source of work that humans direct agents toward. Observability is no longer only for humans to read. When an alert fires or an anomaly is detected, a human can now immediately send an agent to investigate and propose a fix.

Typical examples:

- → **Alerts from Sentry that a developer routes to an agent for diagnosis**
- → **Anomaly detection (cost spikes, latency shifts) that a developer asks an agent to investigate**
- → **Recurring operational issues that a developer assigns to an agent to generate background PRs**

Note: at Level 2, the human remains the trigger. The observed system path doesn't yet feed agents directly. That shift happens at Level 3, when agents begin responding to signals autonomously.

Level 3: Humans as orchestrators

Level 2 introduced agents as participants in the production system. They could generate pull requests, iterate through validation loops, and react to operational signals. Humans remained on the loop, reviewing the evidence produced by deterministic checks and deciding which candidate changes should progress.

In Level 3 the role of the human shifts again.

Instead of supervising individual changes, humans increasingly orchestrate the overall system of agents. The production system begins to run continuously in the background. Agents execute long-running paths and respond to signals from the system, improving software incrementally without waiting for a human to initiate each step.

It's worth noting that very few organizations have Level 3 platforms running already, although they do exist, especially for frontend changes. The difference between Level 2 and Level 3 is therefore not simply more automation. It's a change in how work is triggered and flows through the platform.

At Level 2, agents respond to discrete signals and produce candidate artifacts for human evaluation. At Level 3, the platform itself begins to operate as a background execution layer where agents continuously propose and refine improvements within predefined guardrails.

Humans move further away from direct intervention. Their role shifts toward defining the rules of the system, including what types of work agents may perform, how validation evidence is interpreted, and under which conditions changes may progress automatically.

The platform becomes less a tool for developers and more an industrial system that developers supervise.

Changes in the feature development value stream (Level 2)

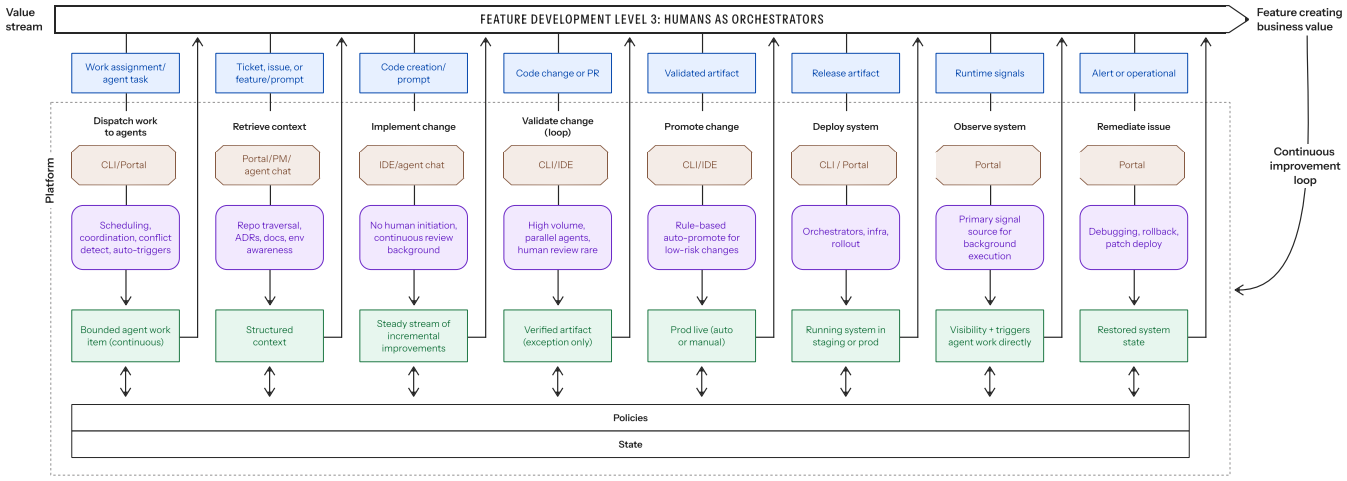
The backbone paths introduced earlier remain the same:

- → **Retrieve context**
- → **Implement change**
- → **Validate change**
- → **Promote change**
- → **Deploy system**
- → **Observe system**
- → **Remediate issue**
- → **Dispatch work to agents**

The structure of the production system, therefore, remains stable. What changes is how frequently these paths are executed and how independently they can operate.

Several paths now run continuously in the background rather than being triggered by individual work items. Agents repeatedly cycle through observation, implementation, and validation as they attempt to improve the system.

FEATURE DEVELOPMENT VALUE STREAM AT LEVEL 3



PATHS EVOLVED FROM LEVEL 2 TO LEVEL 3

Altered New Unchanged

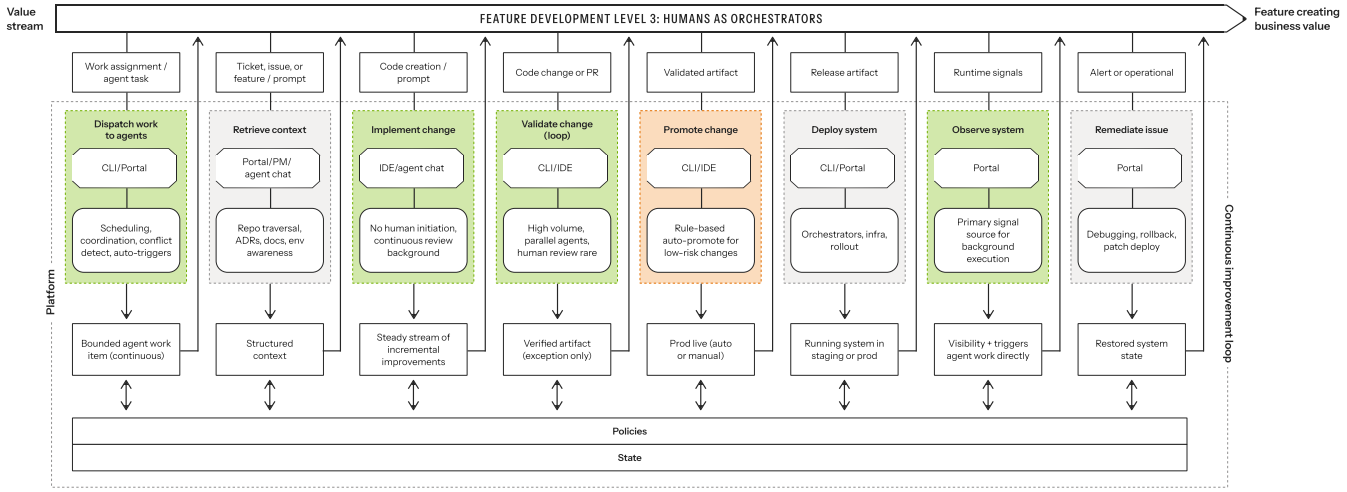


Figure 11 Level 3: Humans as orchestrators - the platform runs continuously in the background, displayed using the path to outcome model with eight paths. Four paths expand significantly: dispatch work gains scheduling and conflict detection, implement change runs without human initiation, validate change handles high volume with rare human review, and observe system becomes the primary signal source for background execution.

Three paths change most significantly.

Dispatch work to agents (expanded)

At Level 2, this path converted signals into agent work items. At Level 3, it evolves into the coordination layer for the entire agent ecosystem.

Work no longer enters the system only through human requests or isolated signals. The platform itself begins to generate work based on patterns observed in the system.

Typical triggers include:

- → **Recurring operational anomalies**
- → **Dependency updates and security advisories**
- → **Performance regressions**
- → **Backlog preparation tasks**
- → **Incomplete work was detected through analysis**

Instead of reacting to individual events, the system now schedules and prioritizes background work streams.

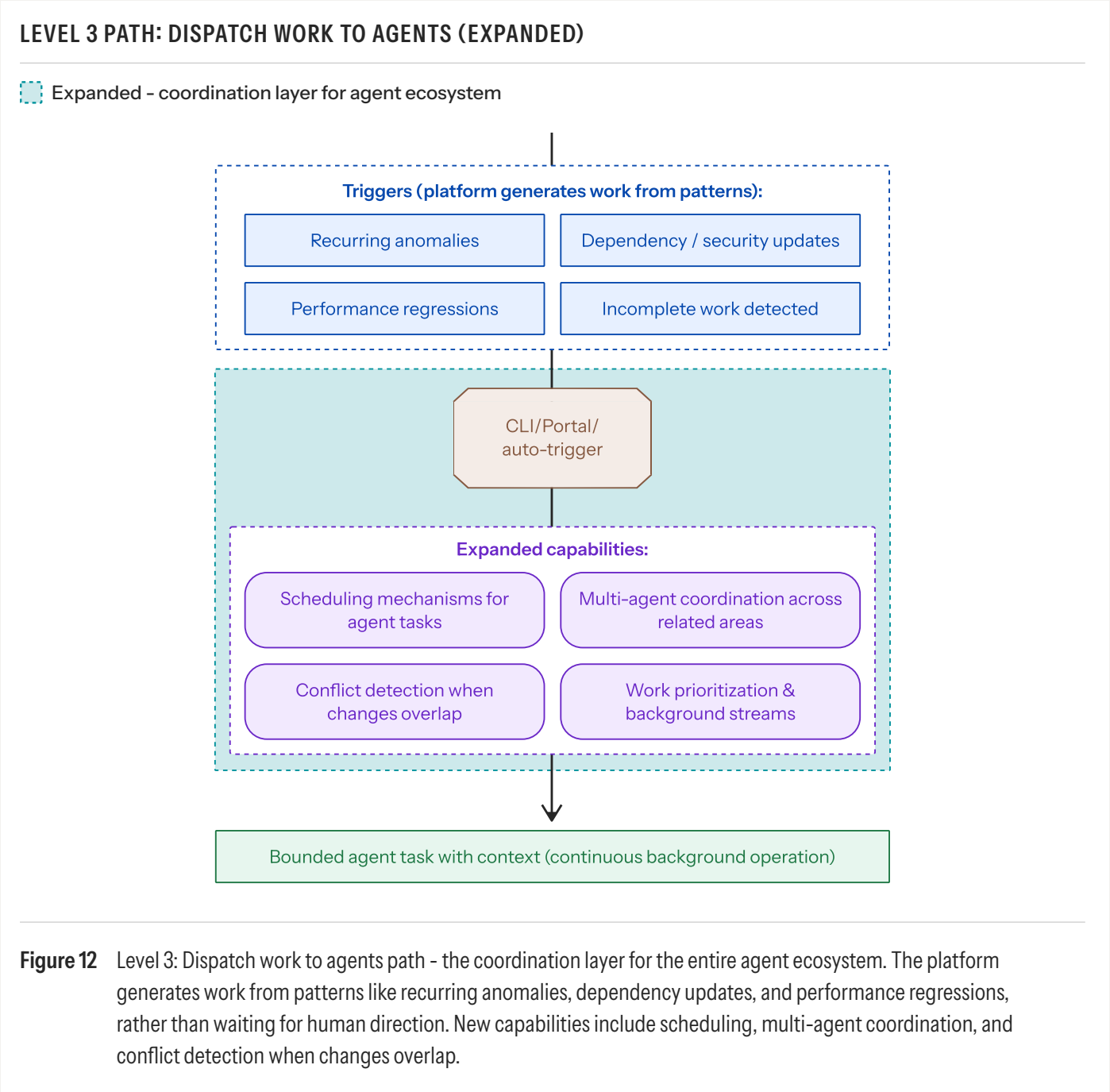


Figure 12 Level 3: Dispatch work to agents path - the coordination layer for the entire agent ecosystem. The platform generates work from patterns like recurring anomalies, dependency updates, and performance regressions, rather than waiting for human direction. New capabilities include scheduling, multi-agent coordination, and conflict detection when changes overlap.

Capabilities inside this path, therefore, expand to include scheduling mechanisms for agent tasks, coordination between multiple agents working on related areas of the system, and conflict detection when changes overlap.

The output remains the same conceptually: a bounded agent task with sufficient context to produce a candidate change.

Implement change (expanded)

At Level 3 agents increasingly implement improvements without explicit human initiation.

Many tasks that previously required manual intervention can now be executed automatically. Examples include dependency upgrades, small refactors, configuration adjustments, and optimizations triggered by system signals.

The artifact produced by this path remains unchanged. Agents still generate candidate changes such as pull requests, configuration patches, or infrastructure updates.

What changes is the cadence and scale. Instead of sporadic developer-initiated commits, the system generates a steady stream of incremental improvements that progress through validation.

Validate change (expanded)

Validation becomes the central control mechanism of the production system. Because agents now operate continuously and often in parallel, the validation layer must handle a significantly larger number of candidate changes.

The validation loops introduced at Level 2, therefore, expand further. Agents repeatedly submit their changes to deterministic verification systems that determine whether the changes satisfy the platform's requirements.

These verification mechanisms typically include:

- → **Functional and scenario testing**
- → **Architectural compliance checks**
- → **Performance validation**
- → **Dependency and security analysis**
- → **Policy enforcement**

When validation fails, the failure signals are routed back to the responsible agent, which modifies the change and retries the path.

Human review becomes increasingly rare. Instead of inspecting individual diffs, humans intervene only when deterministic validation produces ambiguous results or when the system proposes high-impact changes.

Promote change (altered)

Promotion evolves again at Level 3. At Level 2, humans still reviewed validation evidence and decided whether a change should move forward. At Level 3, many low-risk changes can be promoted automatically once sufficient validation evidence exists.

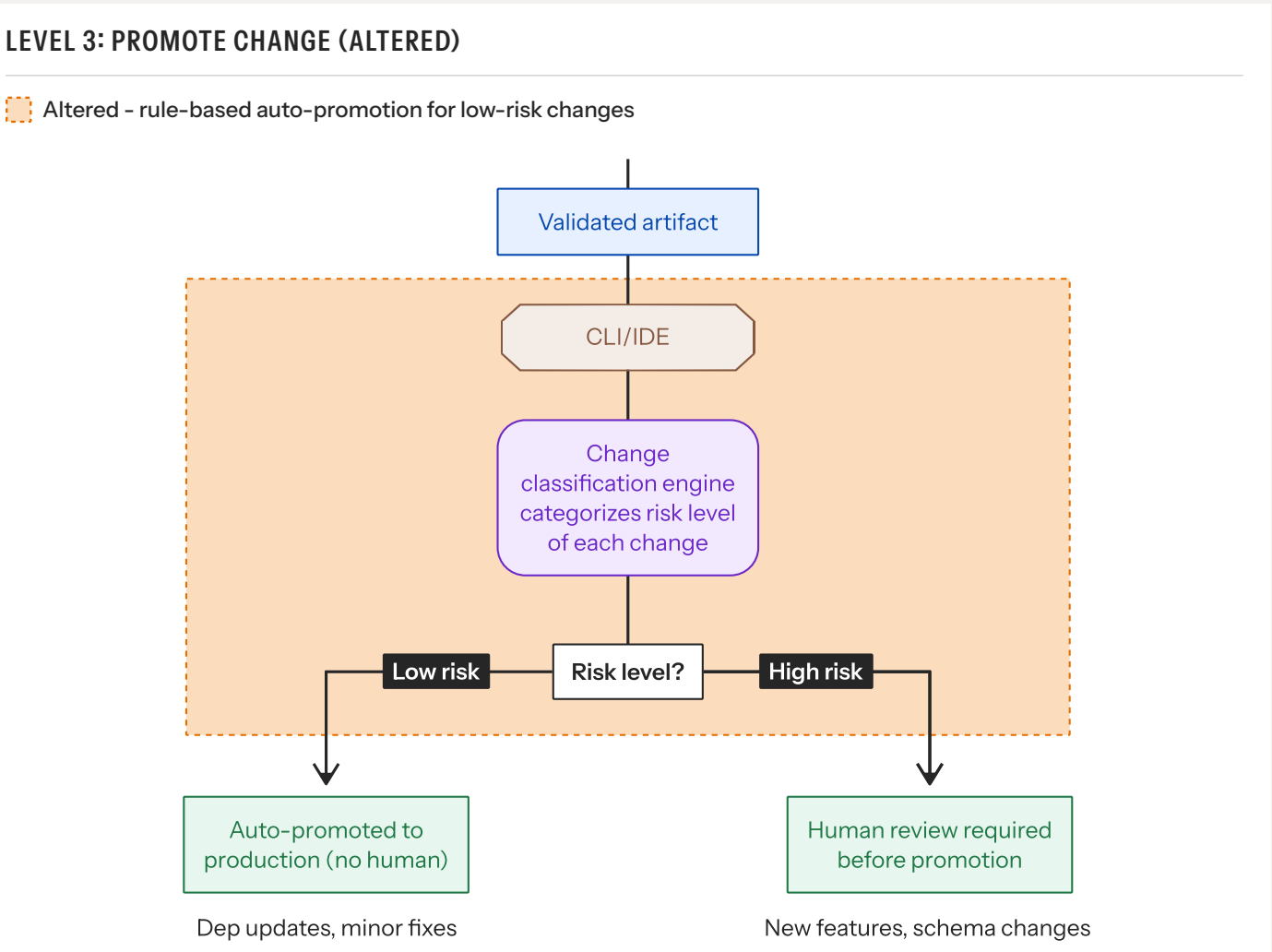


Figure 13 Level 3: Promote change path - rule-based auto-promotion for low-risk changes. A classification engine categorizes each change by risk level. In low-risk changes, dependency updates go straight to production, while high-risk changes still require human review. The key shift: humans review the rules, not every individual change.

Promotion decisions therefore become rule-based. Examples of changes that may progress automatically include dependency updates, minor performance improvements, and fixes for well-understood operational issues.

Humans remain responsible for defining the promotion rules and reviewing exceptional cases. Their role shifts from approving individual changes to designing the policies that determine when the system may advance work independently.

Observe system (expanded)

Observation becomes the primary signal source for the background execution layer.

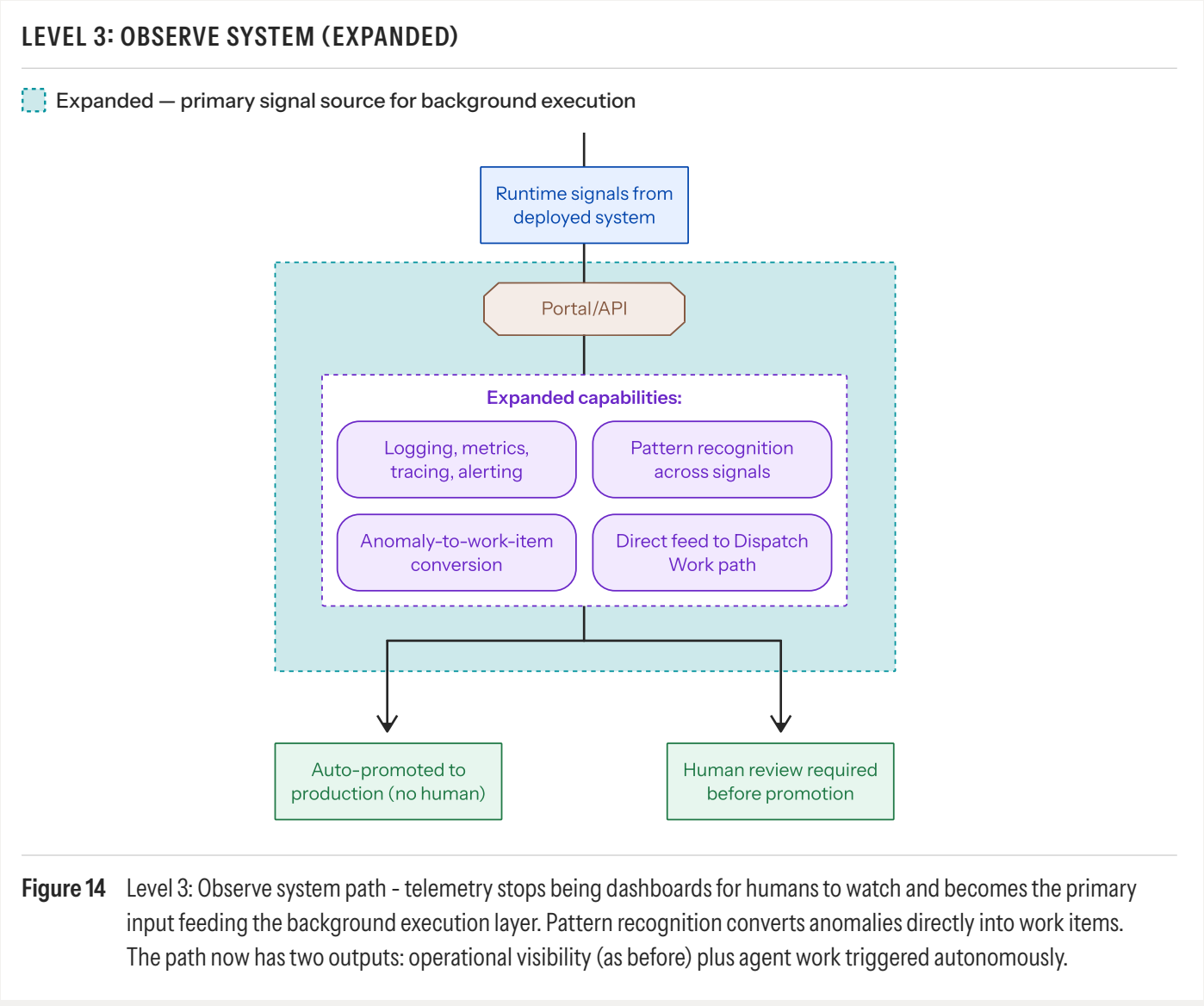


Figure 14 Level 3: Observe system path - telemetry stops being dashboards for humans to watch and becomes the primary input feeding the background execution layer. Pattern recognition converts anomalies directly into work items. The path now has two outputs: operational visibility (as before) plus agent work triggered autonomously.

Operational telemetry increasingly triggers agent work. Signals such as error spikes, latency changes, or cost anomalies can initiate new tasks through the dispatch work to agents path.

In this sense, the observability layer becomes the sensing system of the platform. It continuously feeds information into the production system, enabling agents to propose improvements without waiting for explicit human requests.

Level 4: Fully autonomous (outlook)

The fourth level represents the logical continuation of the patterns described in the previous sections.

At Level 3, agents already execute large parts of the production system in the background. They retrieve context, implement changes, validate results, and propose improvements based on signals from the platform. Humans supervise the system and define the policies that determine how work progresses. Level 4 pushes this model further.

Instead of waiting for humans to dispatch work or supervise individual improvements, the production system itself becomes autonomous for defined classes of work. Agents continuously monitor the environment, initiate new paths when necessary, and evolve software inside the boundaries defined by the platform.

It's important to acknowledge that there are still very few publicly documented examples of organizations operating software production systems at this level. While we've encountered isolated cases of highly autonomous workflows, most teams today operate somewhere between Level 1 and Level 2, with some treading the edge of Level 3.

Level 4 should therefore be understood less as a widely adopted industry practice and more as the logical endpoint of the trajectory we're observing. The assumptions described in this section are based on patterns emerging in advanced agentic workflows and on the architectural constraints required to make those workflows safe.

The key difference from previous levels is simple. Agents no longer wait for explicit work assignments. They initiate work themselves in response to environmental signals. One example we've seen is a SaaS company where agents listen to customer calls and transcribe them. When frontend bugs are mentioned, the system automatically runs tests, detects the issue, and fixes it.

Customer behavior, operational telemetry, cost anomalies, and security alerts can all trigger new work items that agents attempt to resolve automatically.

Seen through the path to outcome model, the production system still consists of the same backbone paths introduced earlier. Retrieve context, implement change, validate change, promote change, deploy system, observe system, remediate issue, and dispatch work to agents all remain part of the system.

What changes is who initiates these paths and how frequently they execute.

What Level 4 unlocks

The implications of a production system that can initiate its own work are significant. Several classes of activity that are prohibitively expensive or slow when humans must trigger every step become continuous background operations.

Competitive feature tracking

An autonomous production system can monitor competitor product surfaces, detect new capabilities, and generate candidate implementations. A Level 3 system requires a human to notice the competitor change, file a ticket, and dispatch an agent. A Level 4 system detects the change, proposes an implementation, validates it against the existing codebase, and surfaces a ready-to-review PR. The human reviews and decides whether to ship, but the detection and implementation cycle happens without human initiation.

Demand-responsive adaptation

When usage patterns shift, whether seasonal load changes, geographic expansion, or unexpected traffic spikes, a Level 4 system can initiate infrastructure scaling, configuration changes, and even feature adjustments in response. Rather than an engineer interpreting a dashboard and filing a change request, the system itself proposes and validates the adaptation. The human governs the policy boundaries within which these adaptations are allowed.

Continuous large-scale refactoring

Refactoring large codebases is one of the most avoided tasks in software engineering because the cost-benefit ratio is unfavorable when humans must execute every step. At Level 4, the production system can continuously identify technical debt, dependency risks, and architectural drift, and generate and validate refactoring changes in the background. Code modernization becomes a continuous process rather than a quarterly initiative that never gets prioritized.

Proactive security and compliance remediation

Instead of responding to vulnerability disclosures with manual patching cycles, a Level 4 system can detect new Common Vulnerabilities and Exposures (CVEs), assess exposure across the codebase, generate patches, validate them through the full testing loop, and present them for promotion. The same applies to compliance drift: policy changes propagate through the system as automated change proposals rather than audit findings.

Self-improving validation

Perhaps the most structurally important capability: the production system can analyze its own failure patterns and improve its operating model. When a class of validation failure recurs, the system can update agent context stacks, adjust testing expectations, or tighten policy rules to prevent the same failure category from reaching validation in the future. The feedback loop that was introduced at Level 2 becomes self-tuning.

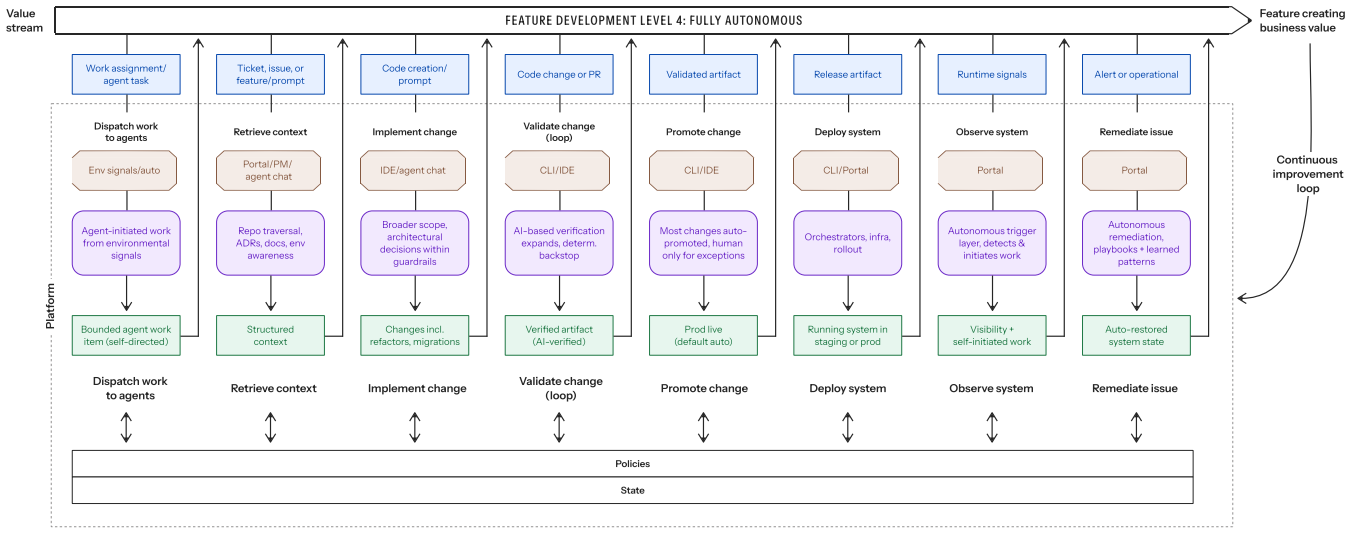
In all of these cases, the underlying pattern is the same. The production system observes a signal, determines that it falls within a governed class of work, initiates the appropriate paths, and presents the result for human review or automatic promotion depending on the risk classification. The human role shifts from initiating and supervising work to defining the policies, boundaries, and risk thresholds that determine what the system is allowed to do on its own.

The platform becomes both a constraint and an enabler. The quality of governance, the precision of policy definitions, and the robustness of guardrails determine how much autonomy can be safely extended. They are also the reasons why Level 4 can't be reached solely by improving agent capabilities. Doing so requires a production system mature enough to govern autonomous execution at scale.

Changes in the feature development value stream (Level 4)

At Level 4, the value stream evolves into a continuous feedback system. Instead of a sequence of human-initiated steps, the system constantly cycles between observation, implementation, validation, and deployment. Signals from the environment trigger work, agents attempt improvements, deterministic systems validate the result, and successful changes propagate back into production.

FEATURE DEVELOPMENT VALUE STREAM AT LEVEL 4



HOW THE PATHS EVOLVED FROM LEVEL 3 TO LEVEL 4

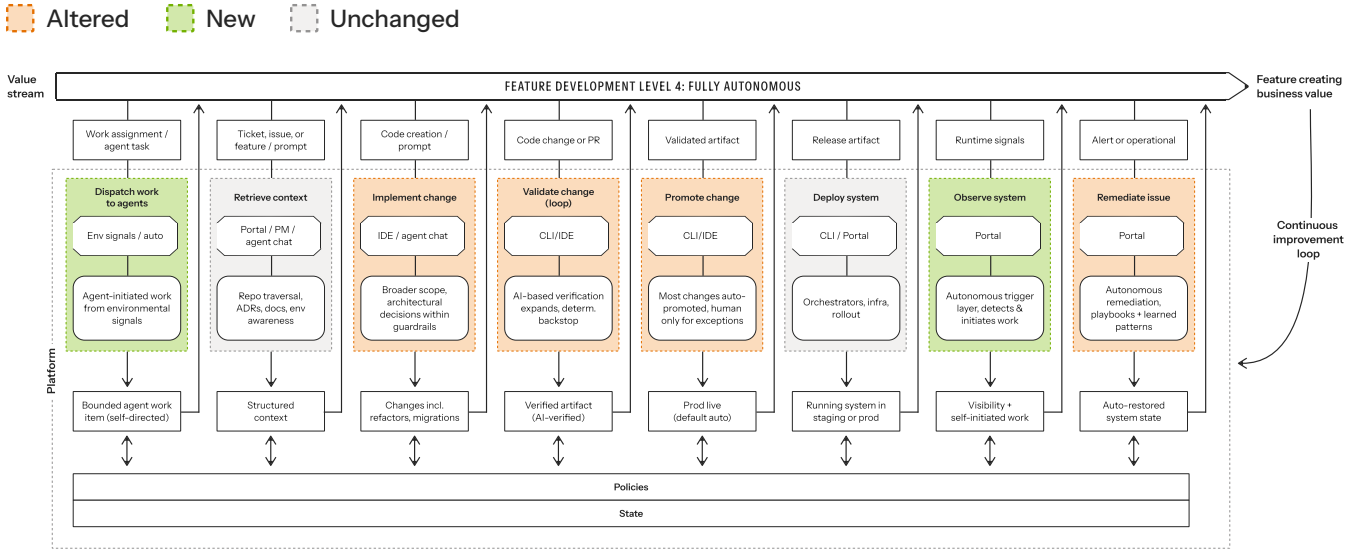


Figure 15 Level 4: Fully autonomous - the system operates with minimal human involvement, as shown in the path-to-outcome model with eight paths. Dispatch work and observe system paths expand to initiate work from environmental signals. Four paths are altered to handle a broader scope, AI-based verification, auto-promotion by default, and self-healing remediation.

Several paths, therefore, change again.

Observe system path (expanded)

Observation becomes the primary entry point for work in the system. In previous levels, observability mainly served human operators or triggered occasional agent tasks. At Level 4 it acts as the sensing layer of the autonomous platform.

Telemetry signals such as error spikes, performance regressions, unusual usage patterns, or cost anomalies continuously feed into the dispatch work to agents path.

The platform, therefore, begins to behave like a feedback system. Observation produces signals, signals generate work, and successful changes alter the behavior of the running system.

Dispatch work to agents path (expanded)

At Level 4, this path becomes the central coordination mechanism for the autonomous platform.

Signals from the environment are continuously converted into agent work items. Instead of waiting for human instructions, the system decides which signals should trigger actionable tasks. Typical triggers may include:

- → **Recurring operational failures**
- → **Dependency vulnerabilities**
- → **Performance degradation**
- → **Infrastructure scaling needs**
- → **Feature opportunities detected through usage patterns**

Capabilities inside this path must support prioritization and conflict resolution while enabling safe scheduling of agent activity. The goal is not unlimited automation but controlled autonomy. The platform must ensure that agents only act within clearly defined boundaries.

Implement change (expanded)

Agents now implement many improvements without explicit human requests.

Work may originate from operational signals, system analysis, or optimization opportunities identified by agents themselves. These changes are still proposed as candidate artifacts, such as pull requests or configuration updates.

Examples include:

- → **Dependency upgrades and security patches**
- → **Performance improvements**
- → **Infrastructure adjustments**
- → **Automated remediation of operational issues**

Although the work is autonomous, the output remains the same: a candidate change that must still pass deterministic validation before progressing.

LEVEL 4: REMEDIATE ISSUE (ALTERED)

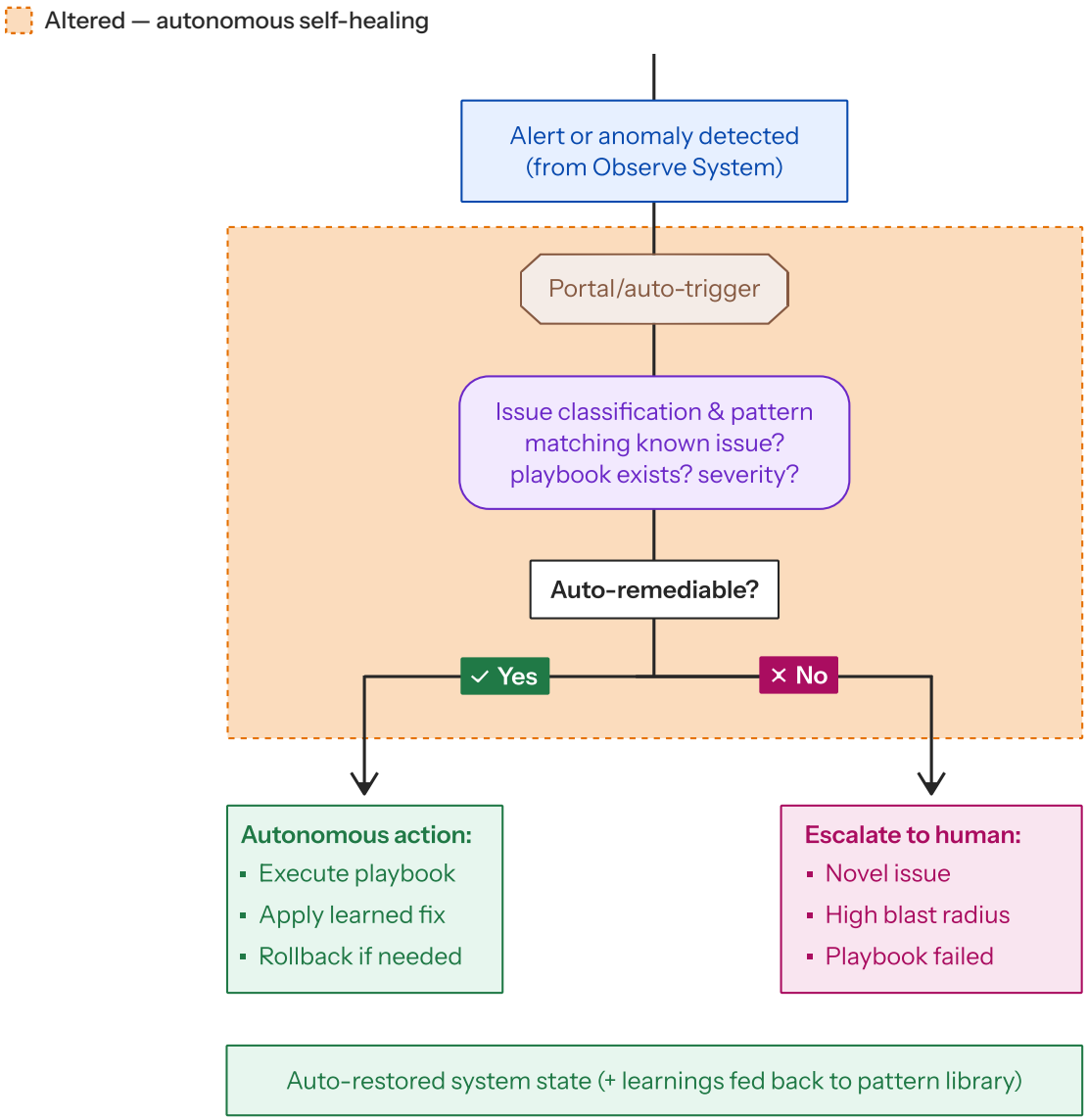


Figure 16 Level 4: Remediate issue path (detail). The system becomes self-healing. When alerts arrive, issue classification matches against known patterns and playbooks. Auto-remediable issues trigger autonomous actions, such as executing a playbook, applying a learned fix, or rolling back, without human involvement. Humans handle only novel or high-blast-radius issues, and every remediation expands the pattern library for future autonomy.

Validate change (critical control layer)

Validation becomes the most important safeguard of the entire production system.

Because agents now initiate changes continuously, the validation layer must ensure that only safe, compliant modifications reach production. Deterministic verification replaces manual approval as the primary control mechanism.

These verification systems typically evaluate:

- → **Functional correctness**
- → **Architectural compliance**
- → **Performance behavior**
- → **Security policies**
- → **Dependency integrity**

Failures are automatically routed back to the agent responsible for the change. The agent attempts to correct the issue and retries the validation path until the change either satisfies the requirements or is discarded.

In this model, validation acts as the industrial safety system that allows autonomous execution to scale.

Promote change (automated)

Promotion decisions become largely automated at Level 4.

If validation provides sufficient evidence that a change is safe and compliant, the platform can automatically advance the artifact. Humans no longer approve individual changes. Instead, they define the rules that determine when promotion is allowed.

Typical automatically promotable changes include routine dependency updates, operational fixes, or small optimizations with clearly defined validation criteria. More complex modifications may still require human escalation, depending on the platform team's policies.

Operating an autonomous production system

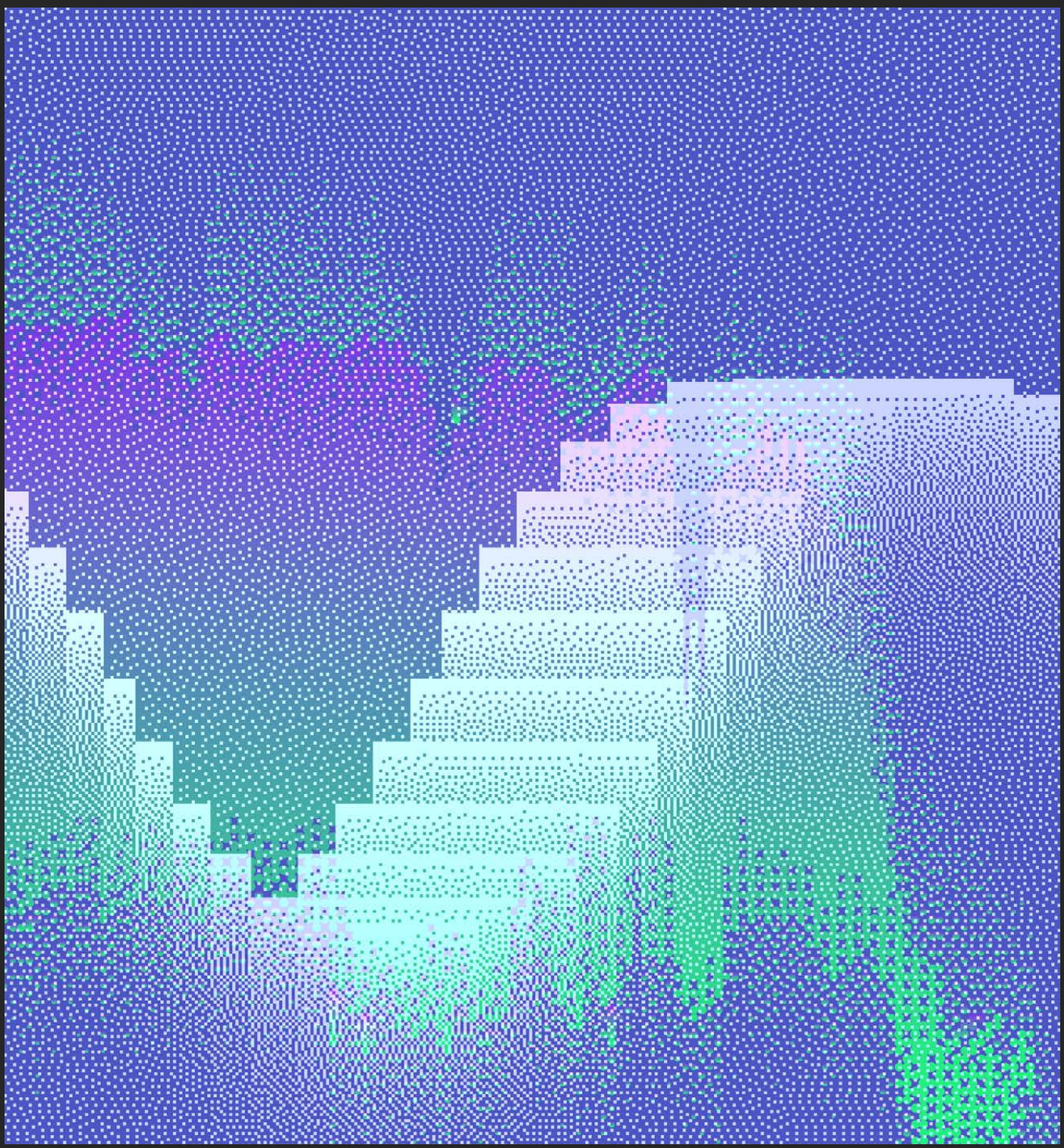
At Level 4, the platform effectively becomes a continuously running production system that evolves software automatically inside predefined guardrails.

Humans don't disappear from the process. Their role changes again. Instead of executing work or reviewing individual changes, they design the rules and constraints that govern the system. Platform engineers define capability boundaries, validation mechanisms, promotion policies, and safety controls that determine how agents operate.

The focus shifts from writing software to designing the production system that generates software. Organizations that reach this stage no longer think of software development primarily as a human workflow. They think of it as an autonomous platform process that continuously adapts software to the environment while remaining bounded by deterministic guardrails.

The decisive factor is how well the platform governs the agent's behavior.

Path change level by level



Path change level by level

As a final overview, the table below shows in detail what paths are added, removed, or altered level by level:

Status

- UNCHANGED — Same as previous level
- EXPANDED — Significantly scaled up
- – — Not yet present
- ALTERED — Meaningfully changed
- NEW — A new path introduced at this level

PATH	LEVEL 0 HUMAN IS THE LOOP	LEVEL 1 HUMAN IN THE LOOP	LEVEL 2 HUMAN ON THE LOOP	LEVEL 3 HUMAN AS ORCHESTRATOR	LEVEL 4 FULLY AUTONOMOUS
Retrieve context	BASELINE Ticket or feature request → repo search, doc retrieval, dependency lookup → context for the developer.	UNCHANGED Still human-driven. Developers can now prompt agents to explain components or summarise files, but the path structure is the same.	ALTERED Primary consumer shifts to agents. Must support agent-scale retrieval: repo/dependency traversal, architectural intent (ADRs, docs), environment awareness. Output is structured context an agent can act on, not just prose.	UNCHANGED Continues to serve agents operating in the background. Reliability of retrieval is critical for continuous execution, but the path structure does not change further.	UNCHANGED Same role as Level 3. Agents retrieve context automatically as part of autonomously initiated work cycles.
Implement change	BASELINE Human writes code manually → editing tools, compilers, dependency managers → candidate commit or code change.	ALTERED Agents assist via prompt-driven code generation and inline explanation. Developer inspects, modifies, and accepts/rejects output. Developer remains the execution engine.	ALTERED Agent-executed for many work items. Human dispatches the agent; agent produces the candidate PR. Work happens in parallel in the background. Multi-repo changes possible but still awkward.	EXPANDED Agents implement improvements without explicit human initiation — dependency upgrades, small refactors, config adjustments, signal-triggered optimisations. Steady stream of incremental background changes rather than sporadic commits.	EXPANDED Fully autonomous. Agents implement changes in response to environment signals (CVEs, anomalies, usage patterns, competitor signals) without human requests. Output is still a candidate artifact (PR, config patch) for deterministic validation.
Validate change	BASELINE Code change or PR → CI pipelines, unit tests, integration tests, security scans → verified artifact or failing validation	UNCHANGED Validation still a single human-walked gate. CI runs on the developer's PR and humans review before progression. Scope of checks may widen slightly but the structure is unchanged.	ALTERED Becomes a feedback loop. Failures route back to the agent to fix and retry. CI capacity pressure rises sharply. Validation widens: scenario execution, performance, dependency/licence checks, security	EXPANDED Central control mechanism of the production system. Must handle a much larger volume of concurrent candidate changes. Human review reserved for ambiguous or high-impact	EXPANDED — CRITICAL CONTROL LAYER The most important safeguard of the entire system. Replaces manual approval as the primary control mechanism. Functional, architectural, performance,

			scanning, IaC policy enforcement (OPA, Kyverno, Checkov). AI-assisted review runs early in the loop. Aspiration: by the time a change reaches validation it should almost always pass.	results; otherwise deterministic checks govern progression.	security, and dependency checks all run. Failures auto-routed back to agent. Validation acts as the industrial safety system enabling autonomous execution at scale.
Promote change	BASELINE Validated artifact → review systems, promotion policies, environment checks → approved artifact ready for deployment.	UNCHANGED Humans still read diffs and approve changes manually before promotion. No structural change to the path.	ALTERED Humans no longer read every diff. Role shifts to verifying system behavior, reviewing evidence (tests, scenarios, policy outcomes), and triaging into "ship now" vs "park with PR attached." Deploy previews used for behavioral verification on frontend.	ALTERED Becomes rule-based. Many low-risk changes (dependency updates, minor perf improvements, well-understood operational fixes) can be promoted automatically when validation evidence is sufficient. Humans define promotion rules rather than approving individual changes.	AUTOMATED Largely automated for safe and compliant changes. Humans define the policies and risk thresholds; the platform advances work within those boundaries. Complex or high-impact changes may still escalate to human review based on policy.
Deploy system	BASELINE Release artifact → deployment orchestrators, infrastructure provisioning, rollout automation → running system in staging or production.	UNCHANGED Same deterministic deployment infrastructure. Agents do not touch this path.	UNCHANGED Controlled delivery systems remain intact. Agents produce artifacts; deployment itself remains a governed deterministic path.	UNCHANGED Execution of deployments continues via same orchestration layer, now at higher cadence as more changes pass promotion automatically.	EXPANDED Deployments can be triggered automatically by the platform in response to autonomously promoted changes, infrastructure scaling needs, or demand-responsive adaptation signals — without human initiation.
Observe system	BASELINE Runtime signals → logging, metrics, tracing, alerting → operational visibility and telemetry for human operators	UNCHANGED Still a passive monitoring layer for human operators. No structural change.	ALTERED Gains importance as a source of work that humans direct agents toward. Humans route Sentry alerts, cost anomalies, and latency shifts to agents for diagnosis and background PRs. At L2 the human remains the trigger — observation does not yet feed agents directly.	EXPANDED Becomes the primary signal source for the background execution layer. Telemetry (error spikes, latency changes, cost anomalies) triggers agent work through the Dispatch path. Observability becomes the "sensing system" of the platform rather than a read-only dashboard.	EXPANDED — PRIMARY ENTRY POINT The main entry point for all work in the autonomous system. Continuously feeds Dispatch Work to Agents. The platform behaves like a feedback system: observation → signals → work items → changes → altered system behavior → new observations.

Remediate Issue	BASELINE Alert or operational signal → debugging tools, rollback mechanisms, patch deployment → restored system state	UNCHANGED Still human-executed. Agents may assist with explanation but the path is human-owned.	ALTERED Humans can now dispatch agents to investigate alerts and generate remediation PRs in the background. Human remains the decision-maker on whether to apply the fix.	EXPANDED Increasingly agent-driven. Recurring operational issues can be assigned to agents to generate and validate fixes as continuous background work, reducing human escalation to exceptional cases.	AUTOMATED Proactive and automated for defined issue classes. Platform detects CVEs, assesses exposure, generates and validates patches, and surfaces them for auto-promotion or human escalation per risk policy — without waiting for manual initiation.
Dispatch Work to Agents	– Not present. All work is initiated and executed by humans directly.	– Humans still read diffs and approve changes manually before promotion. No structural change to the path.	NEW Converts human-directed assignments (ticket, Slack thread, screenshot, Sentry alert) into agent-executable work items with packaged context, non-human identity, workspace provisioning, and output routing. Enables the first real parallelisation: multiple PRs advancing simultaneously.	EXPANDED Evolves into the coordination layer for the entire agent ecosystem. Work is now also generated by the platform itself (patterns, anomalies, dependency advisories). Adds scheduling, multi-agent coordination, and conflict detection for overlapping changes.	

Outlook on our research

Our ongoing research throughout Q2 and Q3 2026 will focus on two areas:

First, we'll publish deeper architectural guidance for ADPs, describing how deterministic platform layers must evolve to safely support agent-driven execution.

Second, we'll provide practical transition playbooks for platform teams moving between levels of agentic development. These guides will focus on the concrete steps organizations can take to redesign their production systems, expand validation capabilities, and safely introduce agents into their software delivery workflows.

As we get closer to Level 4 and fully autonomous systems, we'll also need to spend more time and effort understanding how agents prioritize and weight incoming feature requests and ideas, based on multi-variate input weighting.

The organizations that succeed in this transition won't necessarily be those with the most advanced models. They will be the ones who redesign their platforms early enough to allow agents to operate safely at scale.

For teams beginning this journey today, the most important question is not which model to adopt, but how to evolve the production system that governs software development.

References

[1] Princeton NLP Group, SWE-bench, SWE-bench Verified leaderboard, accessed April 2026. See also: Anthropic, Introducing Claude 3.7 Sonnet, February 2026.

[2] Becker, J., Rush, N., Cunningham, T., Rein, D., & Mahamud, K., We are Changing our Developer Productivity Experiment Design, February 2026.

[3] McKinsey, State of Organizations 2026, February 2026.

[4] Weave Intelligence, Reference architecture of an Internal Developer Platform on GCP, November 2026.

About the authors

Luca Galante

Managing Director, Weave Intelligence

Luca Galante is managing director at Weave Intelligence, the research arm of platformengineering.org, the world's largest platform engineering community with over 280,000 members. He conducts ongoing primary research across hundreds of engineering organizations, distilling patterns from real-world platform setups into authoritative analysis and benchmarks for the industry. He is the host of PlatformCon, the world's largest platform engineering event, and writes to over 100,000 engineers every Friday in his newsletter, Platform Weekly.

Kaspar von Grünberg

Author

Kaspar von Grünberg is an early pioneer in platform engineering. Over the last decade he has been building Internal Developer Platforms (IDPs) at scale, and is responsible for coining the term IDP. A regular speaker on the topic of platform engineering, Kaspar is the author of several associated defining articles.

Mallory Haigh

Head of Platform Education + Advocacy, Platform Engineering

Mallory is a platform engineering leader and educator focused on the intersection of platforms, behaviour, and agentic development. Drawing on experience across full-stack engineering, engineering management, DevRel, and technical customer success, she advances platform engineering as a discipline centred on adoption and real-world outcomes.

Ajay Chankramath

CTO, Platform Engineering Advisory & Consulting

Ajay Chankramath is CTO at Platform Engineering Advisory & Consulting with a career spanning more than three decades in technology leadership, including roles as Head of Platform Engineering at Thoughtworks, CTO at Brillio and CEO at Platformetrics and VP of Software Development at Broadridge. His research covers agentic artificial intelligence (AI) in platform engineering, including AI-enabled platforms and platforms purpose-built for agentic coding workflows, as well as platform security spanning zero trust architecture, policy as code, and runtime threat detection. He is the author of Effective Platform Engineering (Manning), Platform Engineer's Handbook (Packt) and Domain Driven Platform Engineering (Springer), among other seminal works.

About Weave Intelligence

Weave Intelligence

Weave Intelligence is a leading analyst firm specializing in platform engineering. By uniting a team of senior analysts with industry experts and enterprise leaders, we deliver the rigorous research that defines the field.

We enable organizations to leverage the #1 trend in IT as the modern framework for operational excellence and innovation.

Weave Intelligence GmbH
Wöhlertstraße 12-13
10115 Berlin

Disclaimer

Weave Intelligence does not endorse any specific vendor, product, or service. The information contained in this report has been obtained from sources believed to be reliable. Weave Intelligence disclaims all warranties as to the accuracy, completeness, or adequacy of such information. This publication is provided on an “as-is” basis without warranty of any kind, either express or implied. Weave Intelligence shall have no liability for errors, omissions, or inadequacies in the information contained herein or for interpretations thereof.