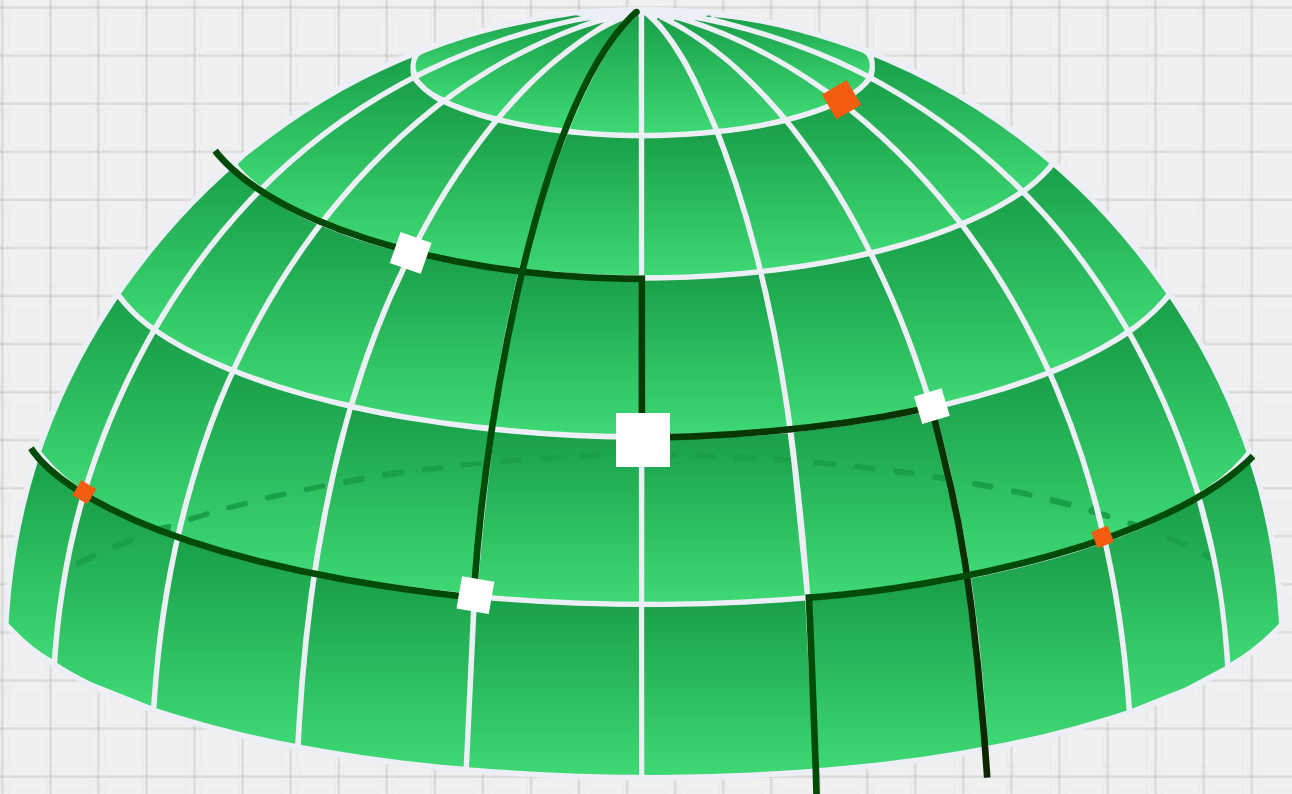


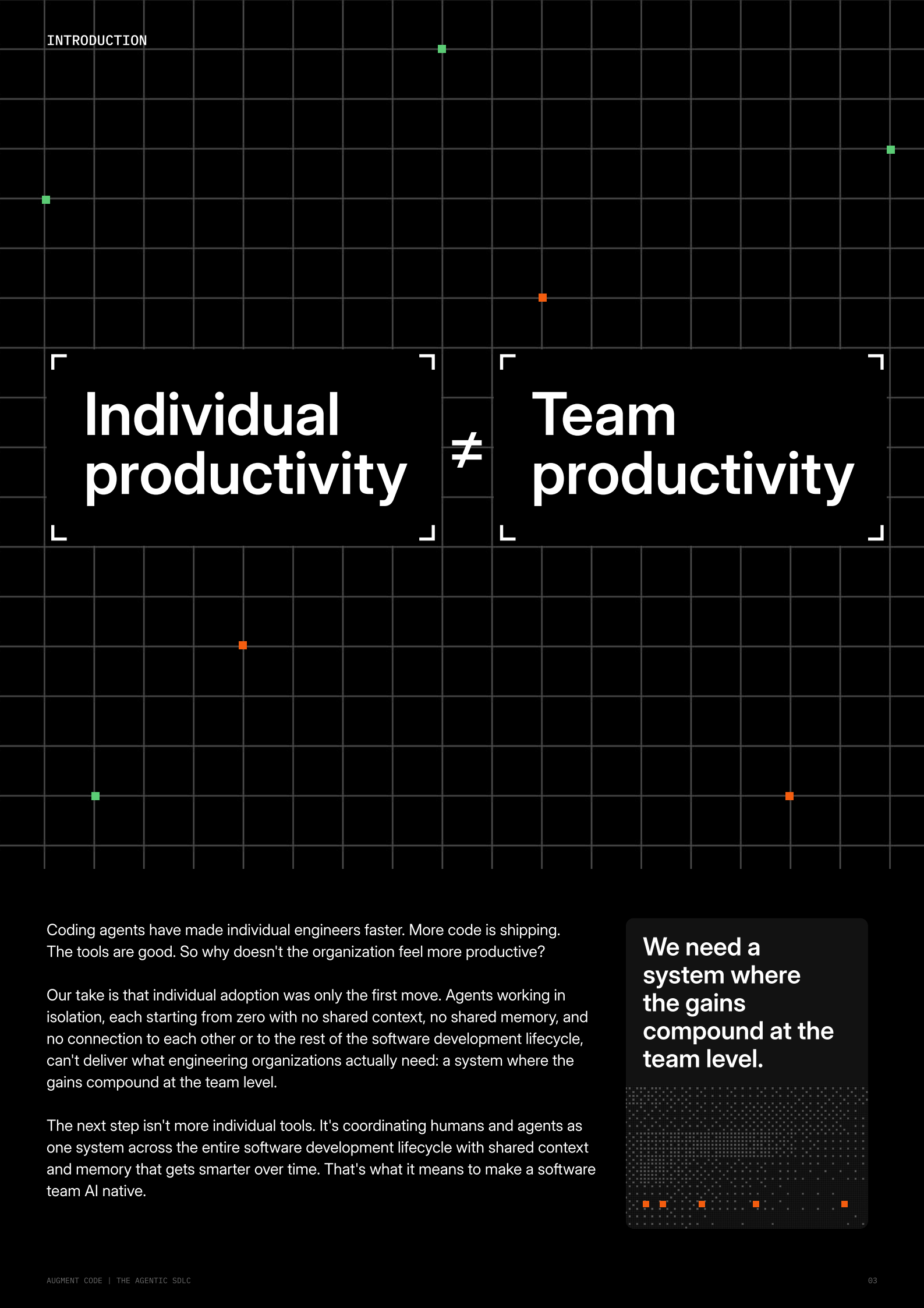
augment code

The Agentic SDLC

■
HOW SOFTWARE TEAMS MOVE FROM
INDIVIDUAL CODING AGENTS TO
A COORDINATED SYSTEM →



- INTRODUCTION 03
- WHERE MOST TEAMS ARE TODAY (AND WHY IT ISN'T ENOUGH) 04
- HOW HUMANS AND AGENTS WORK TOGETHER ACROSS THE SDLC 05
- THE STACK THAT MAKES IT POSSIBLE 08
- THREE PROPERTIES OF A SYSTEM THAT COMPOUNDS 09
- THE SHIFT FROM INDIVIDUAL CODING AGENT ADOPTION TO A COORDINATED, TEAM-LEVEL SYSTEM 10



Individual
productivity

≠

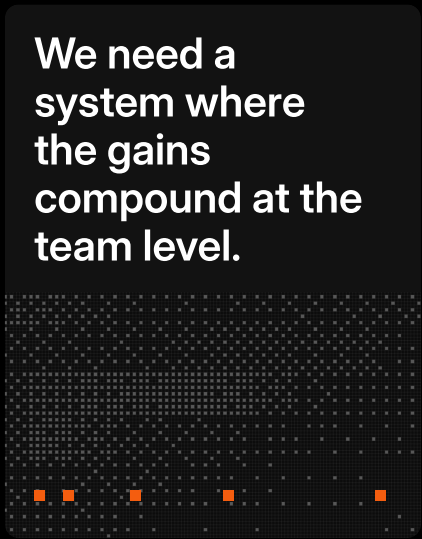
Team
productivity

Coding agents have made individual engineers faster. More code is shipping. The tools are good. So why doesn't the organization feel more productive?

Our take is that individual adoption was only the first move. Agents working in isolation, each starting from zero with no shared context, no shared memory, and no connection to each other or to the rest of the software development lifecycle, can't deliver what engineering organizations actually need: a system where the gains compound at the team level.

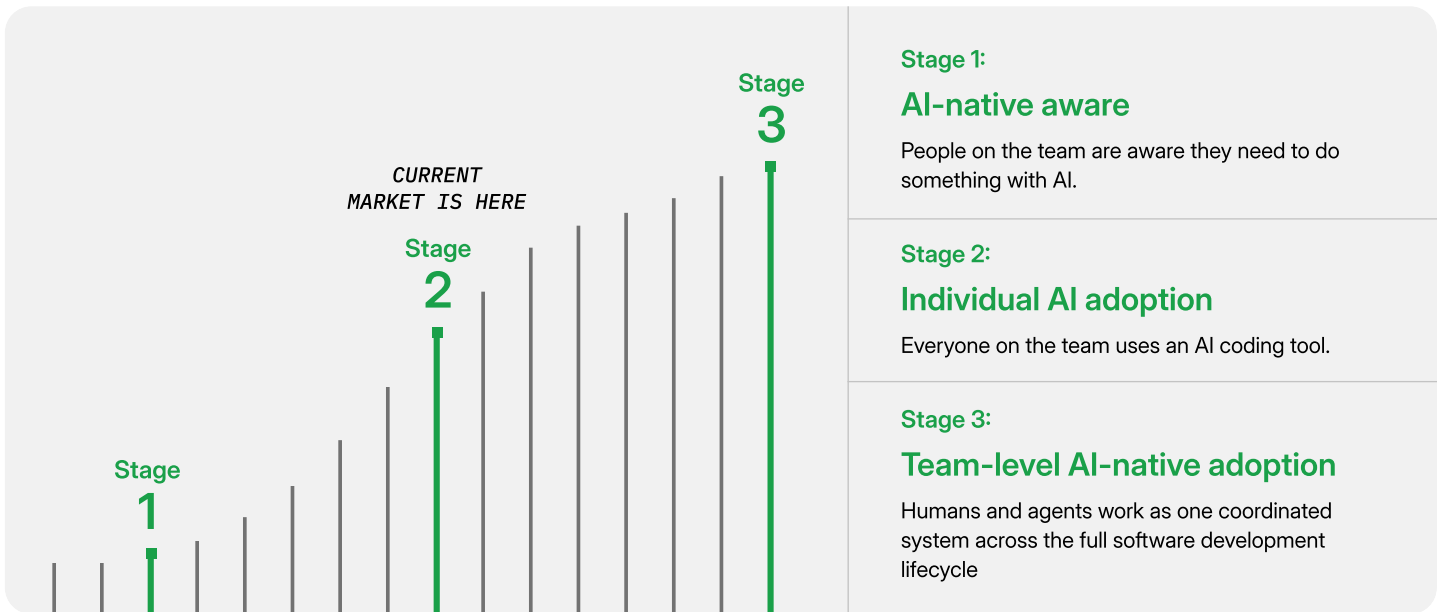
The next step isn't more individual tools. It's coordinating humans and agents as one system across the entire software development lifecycle with shared context and memory that gets smarter over time. That's what it means to make a software team AI native.

**We need a
system where
the gains
compound at the
team level.**



Where most teams are today (and why it isn't enough)

Most engineering leaders equate AI native with individual adoption: everyone on the team is using a coding agent. That's a real milestone. But it's stage two of three.



The organizations stuck at Stage 2 share a common set of symptoms: engineers buried in agent-generated PRs, quality slipping, 2am alerts up, and a growing gap between what was promised and what's showing up in org-level outcomes. The productivity gains stayed with the individual. Nothing compounded.

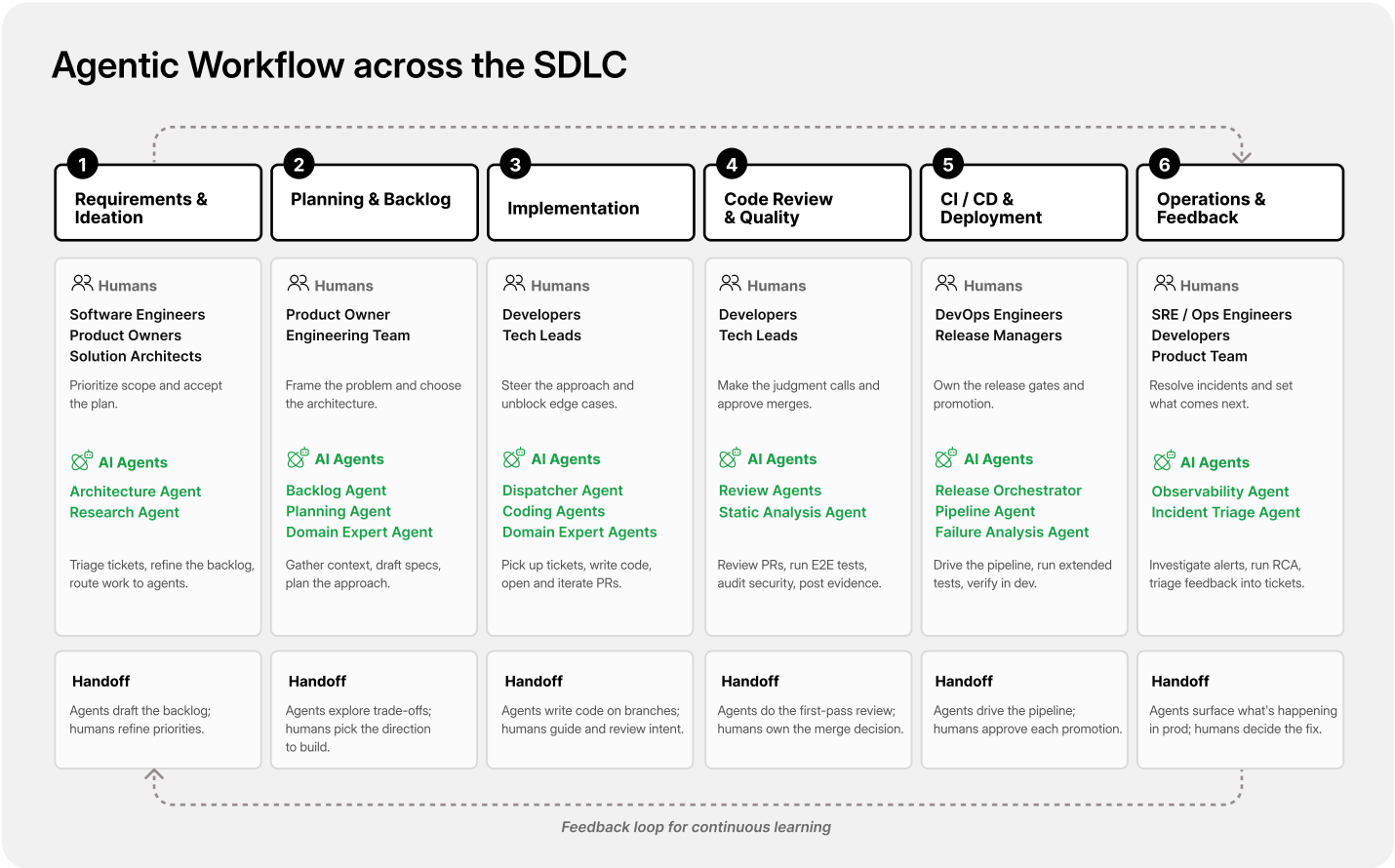
What's missing isn't more licenses. It's a system where humans, agents, code, tools, and memory work together at the org level. One system, not a swarm of disconnected workers.

How humans and agents work together across the SDLC

Team-level AI native doesn't mean agents do everything. It means humans and agents each do what they're best at across every stage of software delivery, not just inside the IDE.

Our philosophy at Augment: humans steer; agents execute. At every stage, there is high-leverage work that belongs to humans (direction, architectural decisions, quality ownership) and execution work that agents can take on at speed and scale.

Humans steer;
agents execute



[0 1] Planning and triage

Planning used to be a human-only activity. Engineers gathered requirements, debated scope, and tried to break work into actionable tasks, often without full visibility into what the codebase would actually need. The result was backlogs full of underspecified tickets and surprises mid-sprint.

Agents can now change what comes into a planning session. They can analyze the existing codebase and architecture to surface dependencies, flag risks, and generate the engineering context that makes work items actionable before a sprint starts. In mature setups, agents observe the backlog continuously and propose routing, matching tasks to the right agent or engineer based on codebase expertise and workload. Humans own priorities and decisions; agents do the preparation work that used to eat planning time.

Humans own priorities and decisions; agents do the prep work.

[0 2] Requirements and design

The traditional gap between what gets documented in a requirements phase and what actually lives in the codebase tends to widen over time. Agents can now help close it. They contribute across multiple lenses simultaneously: security constraints, platform standards, operational requirements, cost implications. Design decisions get stress-tested against what already exists before anyone writes a line of code.

The more important shift is structural. Design discussions stop being isolated, one-on-one conversations with an AI tool and become shared engineering discussions where humans and agents participate together. That visibility matters. Decisions stay legible to the whole team rather than living in one engineer's chat history.

Decisions stay legible to the whole team.

[0 3] Implementation

This is where human-agent collaboration is most intensive, and where the failure mode is most common. Agents can now implement features, generate tests, and execute multi-step workflows. Whether that works in practice comes down to one thing: context. An agent with access to the team's actual codebase, coding standards, and architectural guidance produces reliable output. An agent without it doesn't.

The investment that unlocks this stage isn't more tooling. It's context infrastructure. Structured repository conventions, shared architectural guidance, and standardized patterns make the codebase legible to an agent at enterprise scale. Teams that build this foundation find that some work items become primarily agent-driven; others remain primarily human-authored. The mix shifts over time as context accumulates.

The investment that unlocks isn't more tooling. It's context infrastructure.

[0 4] Code review and verification

Code review is historically a bottleneck. A small number of engineers review everything, catch what they can, and slow down as volume increases. Agents don't just speed up that review; they change the shape of it.

Rather than one generic check, multiple specialized agents can now run review passes in parallel: security, performance, architecture compliance, observability compliance, API governance. Each focused on its domain. They also integrate with the quality tooling teams already run, interpreting CI pipeline failures, and surfacing root-cause analysis rather than raw logs. Engineers aren't triaging; they're deciding. One principle holds across every agentic implementation: verification quality has to exceed generation speed. As agents write more code faster, the review layer has to keep pace.

Engineers aren't triaging; they're deciding.

[0 5] CI/CD and deployment

Deployment has always been the stage where problems that should have been caught earlier become expensive. Agents can now move the catch point upstream. They validate infrastructure definitions before anything ships, flagging security issues, policy violations, cost concerns, and architectural inconsistencies at the point where they're cheapest to fix.

Progressive rollout strategies like feature flags, canary deployments, and blue/green releases also become easier to enforce consistently when agents are part of the deployment workflow rather than relying on engineers to apply them correctly under pressure.

Agents can now move the catch point upstream.

[0 6] Observability and feedback

Operations has historically been isolated from development. Incidents get resolved; the learnings rarely make it back into the backlog in any systematic way. This is the stage where the compounding effect of a team-level system becomes most visible, and where the absence of one is most costly.

Agents can now monitor production continuously, correlate deployment events with anomalies, and surface incidents with context (affected systems, likely cause, estimated blast radius) rather than raw alerts. But the more important shift is what happens next: operational findings flow back into the engineering workflow automatically. Technical debt items, observability gaps, recurring patterns, resilience improvements get created, tracked, and addressed rather than disappearing between sprints. The feedback loop from production to backlog closes. That's where the system starts to compound.

Operational findings flow back automatically.

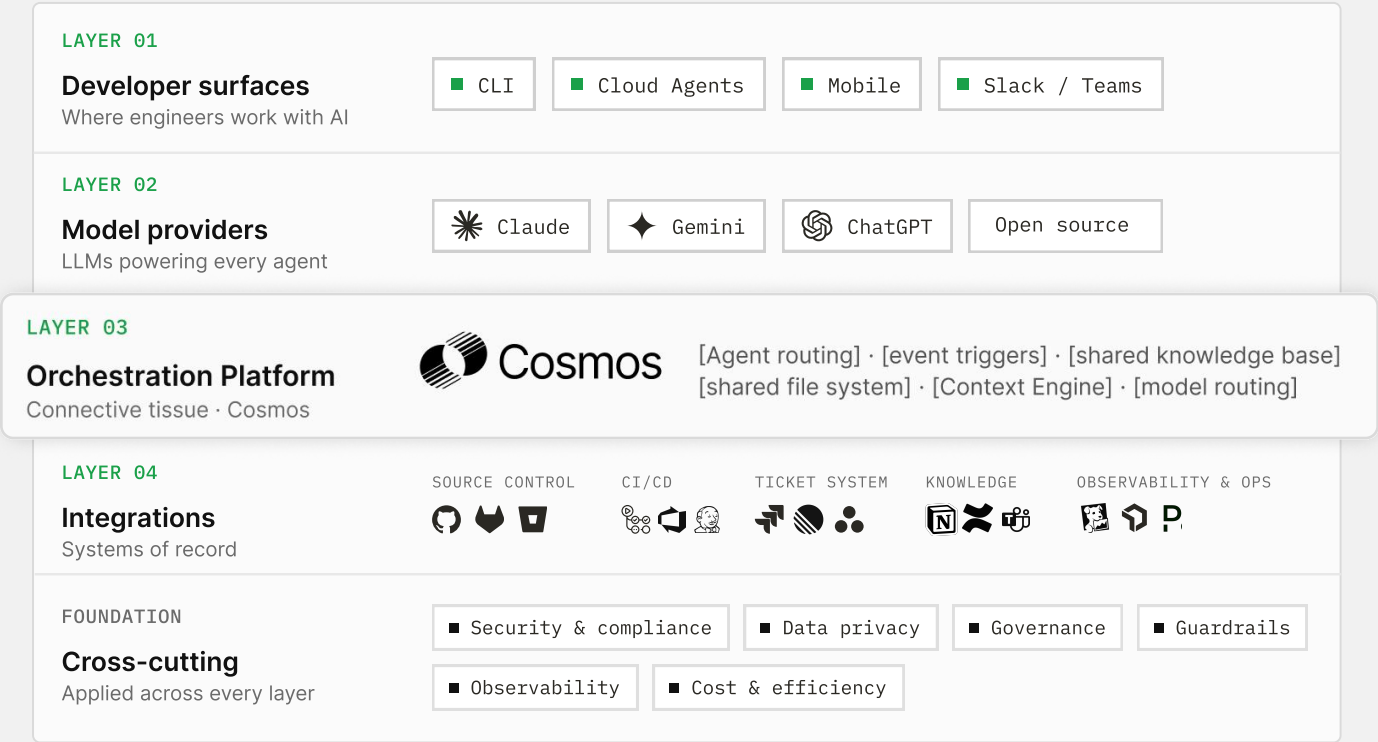
The stack that makes it possible

Running agents across the software development lifecycle requires more than a collection of individual tools. It requires a layered platform where each layer does a specific job, and where the layers connect as one system.

Most teams trying to build this today are stitching point solutions together: one tool for review, another for coding, separate tools for triage and testing.

Each has its own auth, its own data model, its own context. Building a workflow that spans more than one stage is a custom integration project every time. That's the problem a unified platform solves.

The Agentic SDLC Tooling Landscape



Developer surfaces Where engineers already work: CLI, Slack, mobile. Agents don't require a separate console when they can meet the team where it already is.

Autonomous agent execution Agents that execute engineering tasks end-to-end implementing features, running tests, investigating issues, handling multi-step workflows across tools and systems.

Orchestration Platform

This is the layer that turns a swarm of disconnected workers into one coordinated system. Cosmos provides the shared context engine, tenant-wide memory, and event-driven runtime that make agents useful at the team level. It routes work to the right agent and model for each task, maintains shared context across every session, and makes every action observable and auditable. Agents built by one engineer can be promoted into shared capabilities the whole org draws on. Patterns and corrections compound over time rather than resetting.

Key capabilities in this layer: workflow orchestration, model routing (BYOK across major providers), context engine, tenant-wide memory, observability.

Integrations

The engineering systems of record your team already runs: source control, issue tracking, CI/CD, artifact repositories, observability tooling. Agents trigger off events here (PRs, tickets, production alerts, deployment events) and write their outputs back here.

Foundation

Governance, security, identity, and compliance that cuts across every layer. Agent permissions are scoped. Every action is auditable. Human-in-the-loop rules are set by the team and enforced by the platform.

Three properties of a system that compounds

Not all agentic setups are equal. The ones that get better over time share three properties.

[01]

Model agnostic by design

Betting on a single model lab means betting your entire agentic strategy on one provider's roadmap, pricing, and ecosystem. A system built for the long term brings your own keys across Anthropic, OpenAI, Bedrock, Vertex, and open-source models, routing work to the right model for each task without rebuilding workflows when the market shifts.

[02]

Shared memory that accumulates

Every agent on the market today starts from zero. A compounding system is different: patterns, corrections, and institutional knowledge persist across sessions and across the team. Agents built by one engineer get promoted into shared capabilities the whole org draws on. The system gets smarter rather than resetting. That's the difference between a tool and a platform. The agents we have today are the worst they'll ever be.

[03]

Governance by default, not by retrofit

The hardest thing to add later is control. Organizations that bake governance in from the start: observable actions, auditable runs, scoped permissions, explicit human-in-the-loop rules are the ones that can scale agent deployment without incident. Teams set the policies; the platform enforces them.

The shift from individual coding agent adoption to a coordinated, team-level system

The teams modernizing fastest, organizations like Stripe, Ramp, and Uber, are building this system themselves. They're defining how humans and agents divide the work, building shared memory across the team, and connecting agents to every stage of the software development lifecycle.

Most engineering organizations don't have the platform team to build it from scratch. That's not a small gap, it's a multi-year investment, and even the most sophisticated internal teams are finding it harder than expected. Augment Cosmos is what those teams would have built, productized.

The shift from individual coding agent adoption to a coordinated, team-level system is the move from Stage 2 to Stage 3. It's not a tool swap. It's a systems redesign. The organizations that get there first won't just be more productive. The results will compound. The goal is that every agent, every correction, and every pattern adds to the system instead of disappearing at the end of a session.

That's what we think making a software team truly AI native means.

The path from Stage 2 to Stage 3 doesn't have to be a multi-year internal build. Augment Cosmos is that system, productized. Shared context and memory that compounds across the team, agents that work across the full software development lifecycle, humans steering and agents doing. The system gets better over time.

**It's not a tool swap.
It's a systems
redesign.**

**Augment Cosmos
is that system,
productized.**

[See how it works →](#)

