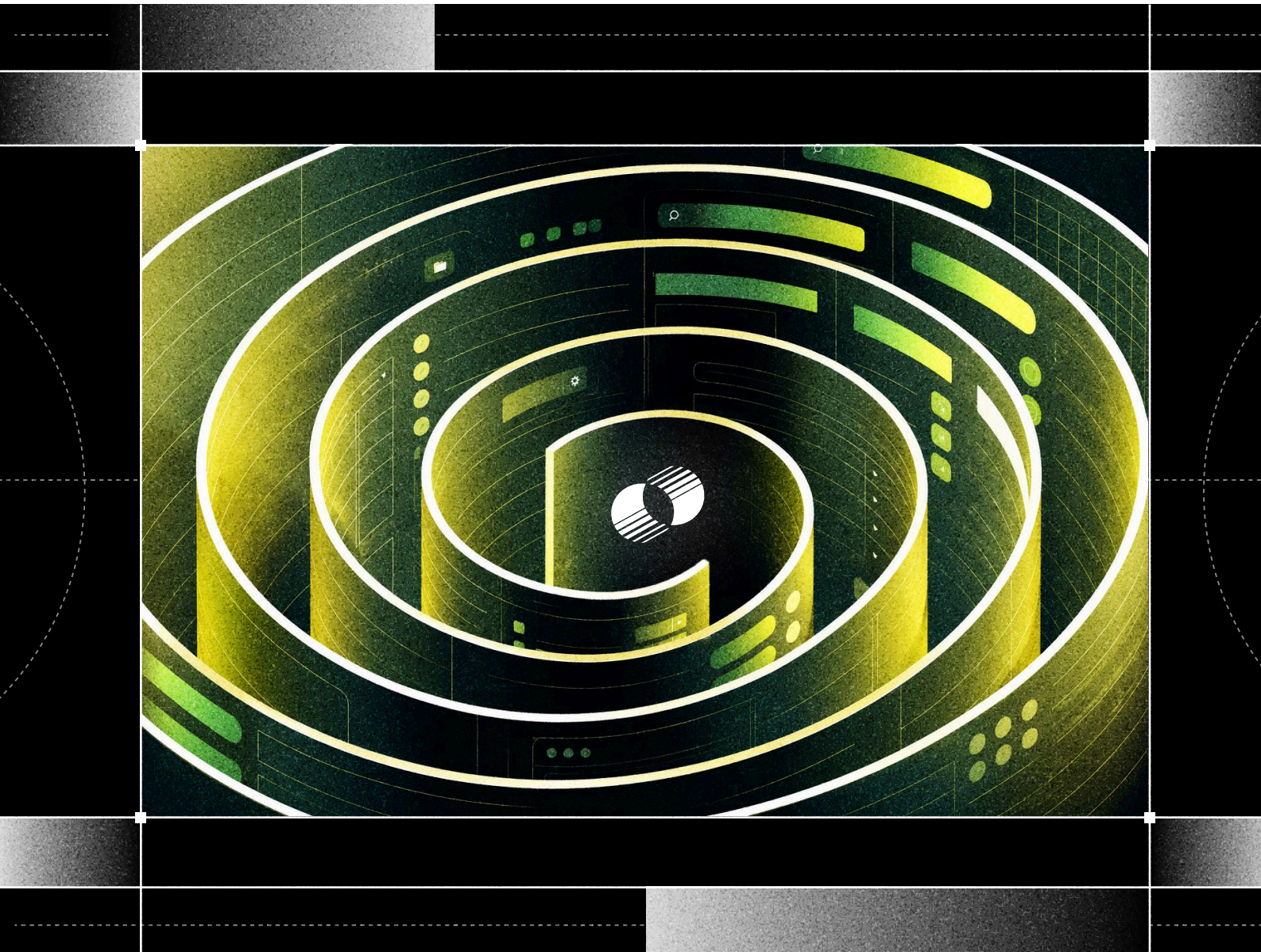
augment code

The engineering leader's guide to building a [software factory]

HOW SOFTWARE TEAMS MOVE FROM INDIVIDUAL CODING AGENTS TO TEAM-LEVEL SOFTWARE DELIVERY



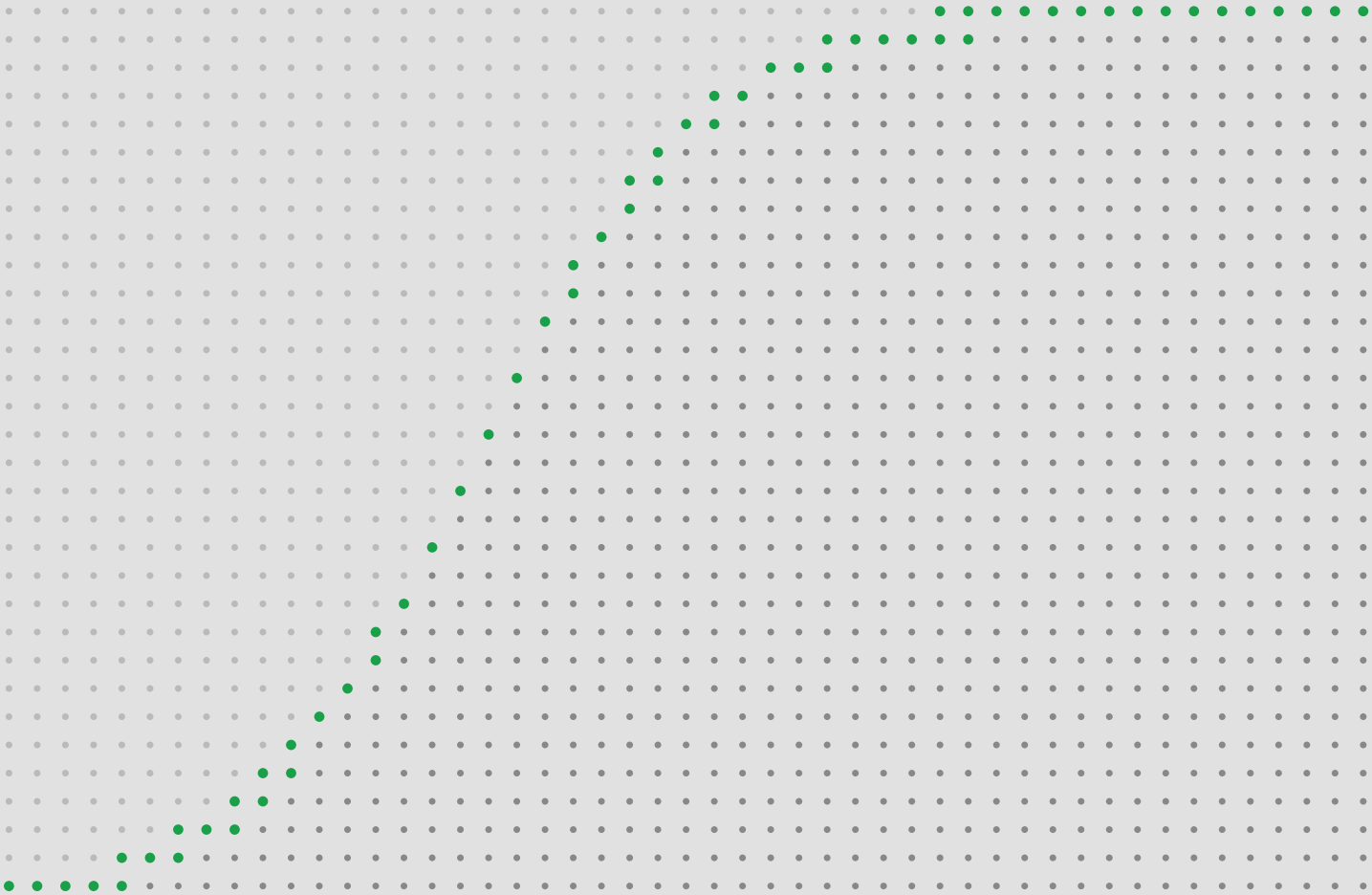
CONTENTS

Why teams are hitting a productivity plateau

In our conversations with hundreds of engineering leaders, we keep hearing the same pattern. Teams expected coding agents to make them 2-3x more productive, but at the organizational level what they report is closer to 20–30%.

PRODUCTIVITY EXPECTATION	PRODUCTIVITY REALITY
2-3x	20-30%

The problem is that individual adoption only changes a single engineer's workflow, not the team's delivery system. Code gets written faster, but it still moves through the same review queues, the same planning cycles, the same deployment process. We call this phase "the productivity plateau" and step-function gains only come when everything around the code speeds up too.



A software factory

Speeding up everything around the code means a system that works at the level of the team, not the individual: one that coordinates humans, agents, context, tools, verification, and feedback across the whole software delivery process. Individual coding agents can't supply that, however good they get, because each one starts from zero and stops at the edge of its own task.

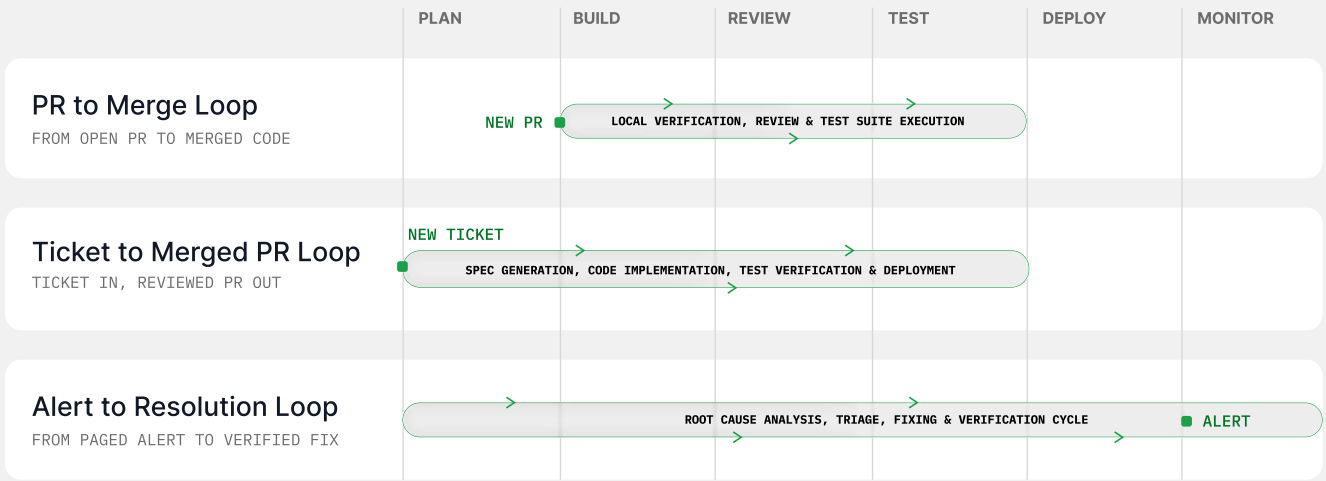
We call this a software factory. Connected, repeatable, quality-gated production that turns intent into verified outcomes, and where each run makes the next one better. A factory isn't just about volume, and definitely not about slop. A good software factory measures throughput in verified outcomes, issues caught by the process, and skilled people spending their time on the highest-leverage judgment calls.

But nobody builds a factory in one go. You build it one production line at a time.

We call these production lines agentic loops. Each one takes a recurring job in the software development lifecycle and handles it end to end. A bug report comes in and a merged fix goes out. A vulnerability gets flagged and it closes. Agents do the repetitive work in the middle, humans step in at the moments that actually need them, and every run leaves the loop better set up for the next one.

When you're running a software factory, you're no longer focused on the individual output of a task, but on designing and optimizing these loops. Where should the human checkpoints sit? How good is your verification? Are the tokens you're burning producing value? This is loop engineering, and it changes what your team measures: outcomes per dollar, not how productive each engineer is with their agent.

Building the software factory



Every loop is a team of specialists

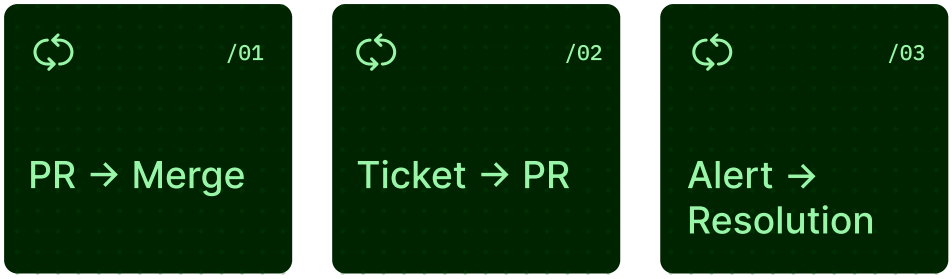
Each loop owns one recurring path from trigger to verified outcome, and runs that path again and again. Build one and you've taken a recurring job off your team's plate. Connect several on a shared foundation and you have a factory.

The thing to understand about a loop is that it isn't one agent working through a checklist. It's several, each with its own objective, tools, and acceptance criteria. One assesses the risk of a change. One does the work. One tries to find fault with it. One decides whether a human needs to see it.

That separation is what makes the output trustworthy. An agent grading its own work will pass its own work. Giving each agent a different objective means every stage has something checking it that wants a different result, which is the same reason you don't let engineers approve their own PRs.

Three loops to get started building your software factory

Across the engineering organizations we work with, the same three loops keep emerging as starting points.



The order you might tackle them in varies, but the pattern is consistent: teams pick the constraint that hurts most, build a loop around it, and the results reshape how they scope the next one. In the following pages we share some of the loop design patterns we see working best.

PR to merge loop

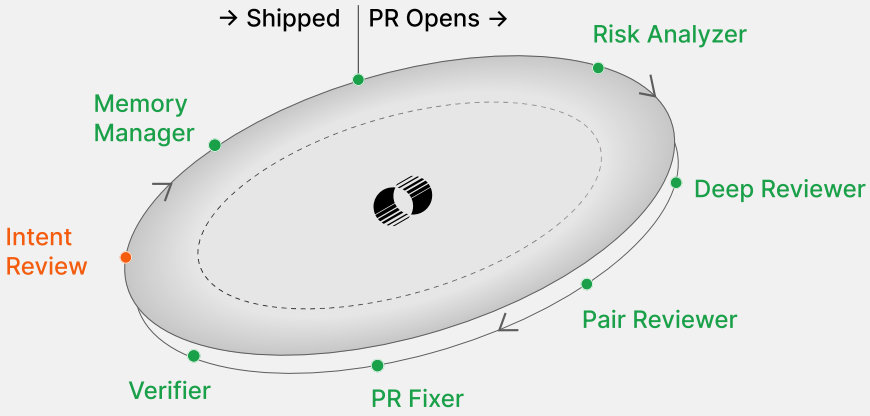
This is the loop most teams need first. When agents write more code, PR volume rises and the constraint becomes confidence. We hit that wall ourselves with 1,400+ PRs open and median time to first human comment around 20 hours.

This loop starts when a PR opens and its goal is to reach a verified merge:

ROLE	ACTION
RISK ANALYZER	Routes the change: auto-approving low-risk PRs and tagging higher-risk ones for human input.
DEEP REVIEWER	Checks correctness line by line; “is there an objective bug?”
PAIR REVIEWER	Surfaces design judgment calls; “does this fit the broader system?” Separating correctness from design judgment matters; one agent doing both will do both poorly.
PR FIXER	Repairs findings, CI failures, and merge conflicts automatically, so most issues resolve without another human round-trip.
VERIFIER	Deploys to an isolated instance, exercises affected behavior, and posts inspectable proof — logs, screenshots, a replayable trace.
INTENT REVIEWER	Makes architecture, intent, and merge decisions.
MEMORY MANAGER	Distills feedback into per-repo knowledge every agent reads next run.

AGENT HUMAN

PR → Merge Loop



66%

FASTER TIME TO MERGE

Ticket to PR loop

Once review is moving, the constraint shifts upstream. The bottleneck is rarely code generation, it's specification, dependency discovery, testing, and coordination. This loop starts when a ticket is assigned and its goal is a merge-ready PR:

ROLE	ACTION
PLANNER	Grounds the ticket in the actual codebase (affected files, dependencies, constraints) and produces a spec a human can review before implementation starts.
CODE AUTHOR	Implements against the approved spec with access to repo context and shared memory.
TEST AUTHOR	Generates and runs tests, catching regressions before the PR opens.
COMPLIANCE CHECKER	Validates against team-defined policies and blocks or flags violations.
INTENT REVIEWER	Reviews the spec before implementation and the PR before merge.

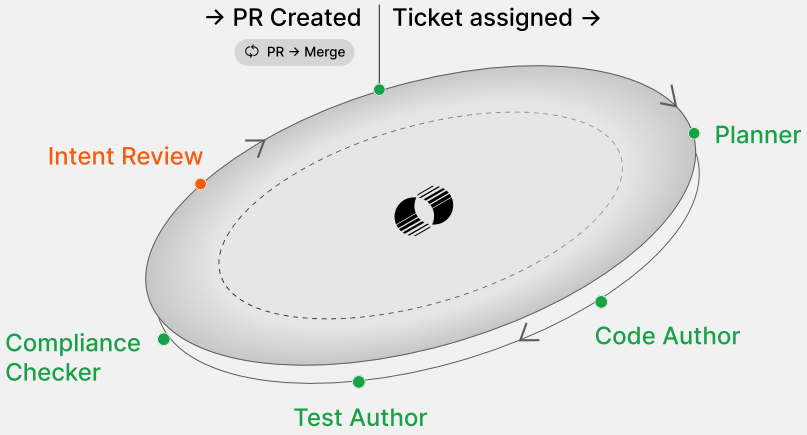
AGENT HUMAN

Once a PR has been created, this loop hands-off to the PR to merge loop above. In one customer pilot: a five-feature epic that originally took a senior engineer about a week was completed in 3.5 hours, including an hour of waiting for human review.

TIME SAVINGS

1 week → 3.5 hours

Ticket → PR
Loop



2-3x

ENGINEERING THROUGHPUT

Alert to resolution loop

On-call time goes mostly to reconstructing context: moving between PagerDuty, Slack, dashboards, and logs to work out what broke. This loop starts when an alert fires and its goal is a verified resolution:

ROLE	ACTION
INCIDENT INVESTIGATOR	Gathers evidence from logs, metrics, recent deploys, and code, then posts an RCA with a recommended remediation before a human has looked.
PR AUTHOR	Creates and manages code fixes when the remediation requires a change, handing the PR through the review loop.
ROLLBACK AGENT	Reverts problematic deployments when that is the faster path.
ESCALATION ROUTER	Identifies the right service owner when the issue is outside agent scope.
INTENT REVIEWER	Scans the RCA, asks follow-ups, and decides whether the remediation is right. Production-impacting calls stay with people.
MEMORY MANAGER	Captures operational learnings so the next investigation starts with more context.

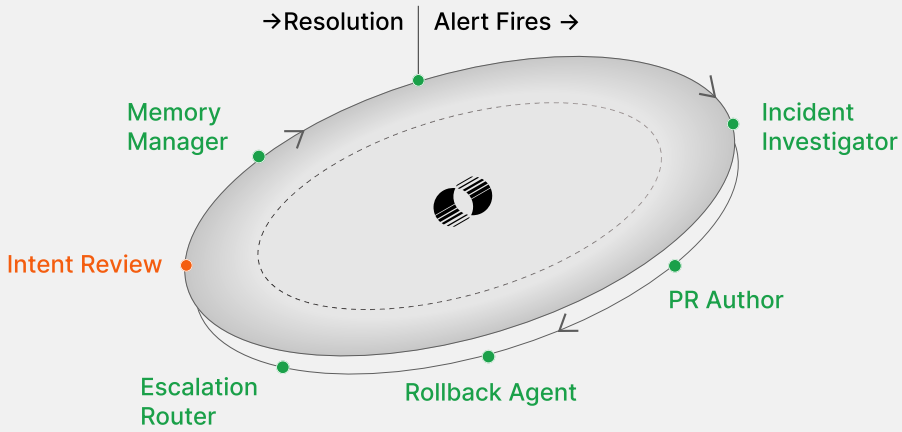
AGENT HUMAN

We ran this across five on-call channels. Agents handled 81.3% of incidents (up from 0.4%). Median time to first RCA fell from 30.1 to 6.2 minutes. On-call engineers merged 44% more PRs per week — time returned to building.

MEDIAN TIME TO FIRST RCA

30.1 mins. → 6.2 mins.

Alert → Resolution Loop



44%

MORE PRS MERGED PER WEEK

The stack to build your software factory on

Individual loops solve individual problems. What turns them into a factory is a shared foundation: the runtime, context, memory, and governance layer that every loop builds on and contributes back to. These are the key components that foundation needs:

Orchestration

The control plane that decides which agent runs next, passes context between stages, handles failures, and enforces sequencing. Teams should be able to tune loop behavior in natural language: adjusting routing rules or adding a step should feel like editing a specification, not shipping a feature.

Cloud runtime and isolated environments

Agents need VMs with real dev tooling where they can check out, build, deploy, and test. Each run needs its own isolated instance so concurrent work does not interfere.

Access to context

Code, tickets, logs, metrics, deploy history, coding standards, runbooks. Every input an agent cannot reach is a class of problem it cannot solve. For large codebases, that means deep codebase understanding, not just file search.

Event-based triggers

Loops start from events: a PR opens, a ticket lands, an alert fires. Agents should subscribe and wait, pausing until a human approves, a CI check passes, or a dependent loop finishes. This is what makes human-in-the-loop control practical at scale.

Shared memory

What one loop learns should be available to every other loop. A correction in review improves future authoring. An incident pattern speeds up the next diagnosis. Without shared memory, each loop operates in isolation and the factory never compounds.

Model routing

No single model is best at every task. Route each to an appropriate model, swap providers as the market moves, and make cost-quality tradeoffs visible.

Governance and audit-ability

Scoped permissions, audit trails, and enforced human-in-the-loop rules. Governance is what lets leaders trust the system enough to widen the autonomy.

PEARL TECHNOLOGIES LIMITED

CHALLENGE

Pearl Technologies runs an engineering team of over 100 people building a transportation management system. Before Cosmos, moving a ticket to a production-ready PR meant manual story writing, local environment setup, and repetitive unit testing. Knowledge transfer needed a scheduled call or a written handoff.

SOLUTION

They rebuilt around cloud-run loops connected to their existing systems. Cosmos integrates with Azure DevOps through webhooks, and the loop writes user stories from high-level instructions, generates code, handles unit testing, and opens PRs. Workspace segregation separated front-end, back-end, and DevOps contexts, and session sharing let one developer hand a full working context to a teammate.

RESULT

In a four-day baseline window in early April, the team completed zero PRs. In the 13 working days after adopting Cosmos, they completed 728. Complex migrations that had spanned multiple days began completing same-day. Efficiency climbed from 3.2 to 4.7 PRs per session over three weeks. Pearl reports the overall change as a 3x productivity gain.

ENGINEERING TEAM

100+

PRS IN 13 DAYS

728

PRS PER SESSION

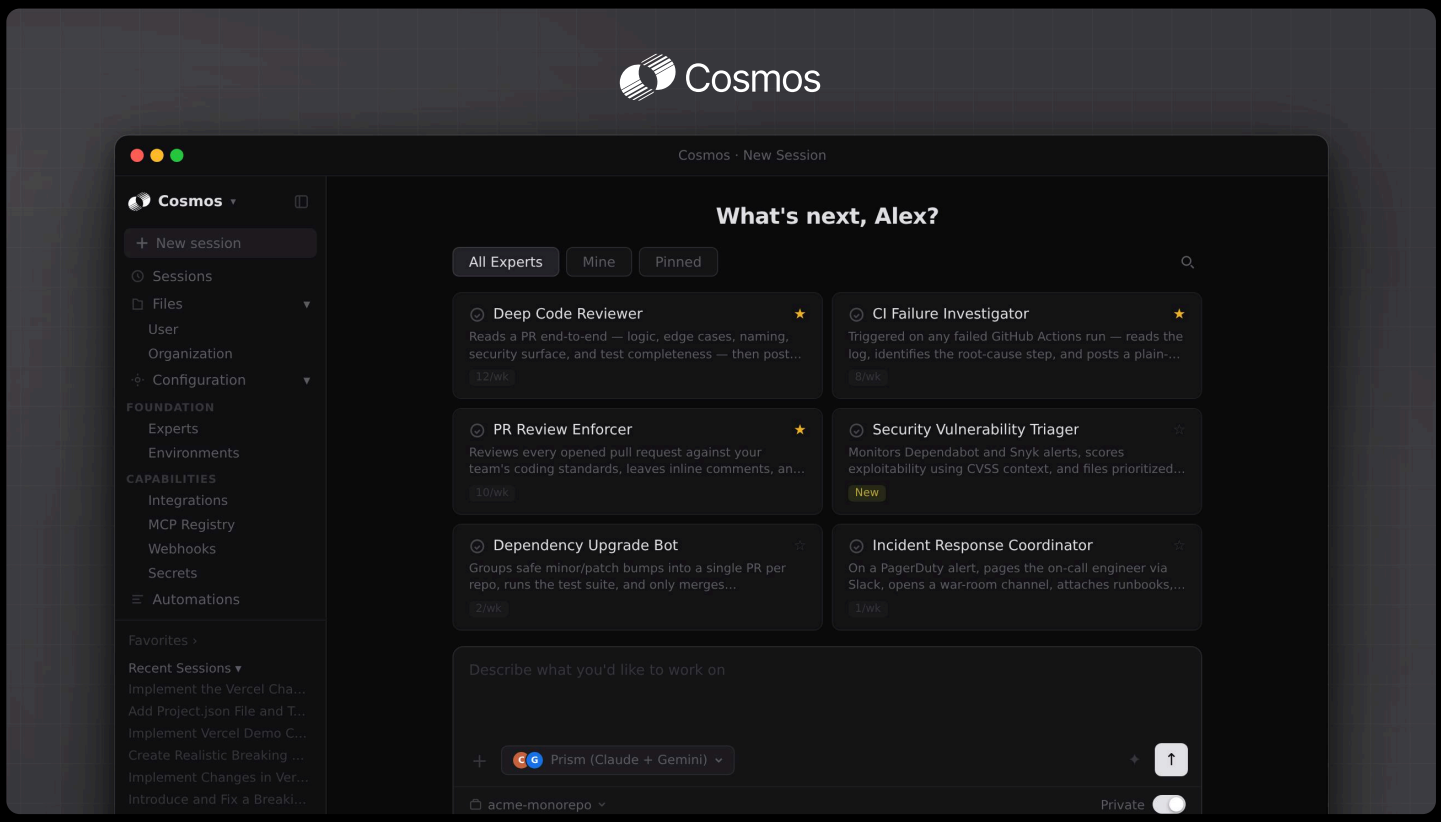
3.2 → 4.7

PRODUCTIVITY GAIN

3x

Get started building your software factory

The productivity plateau breaks when you stop optimizing individual engineers and start optimizing the system they work in. Pick the place where work waits longest in your organization. Build a loop around it. That's your starting point. Every loop you add after compounds on the same foundation.



Build our first loop with Cosmos →

Book a working session to scope with our team →

augment code

AUGMENTCODE.COM