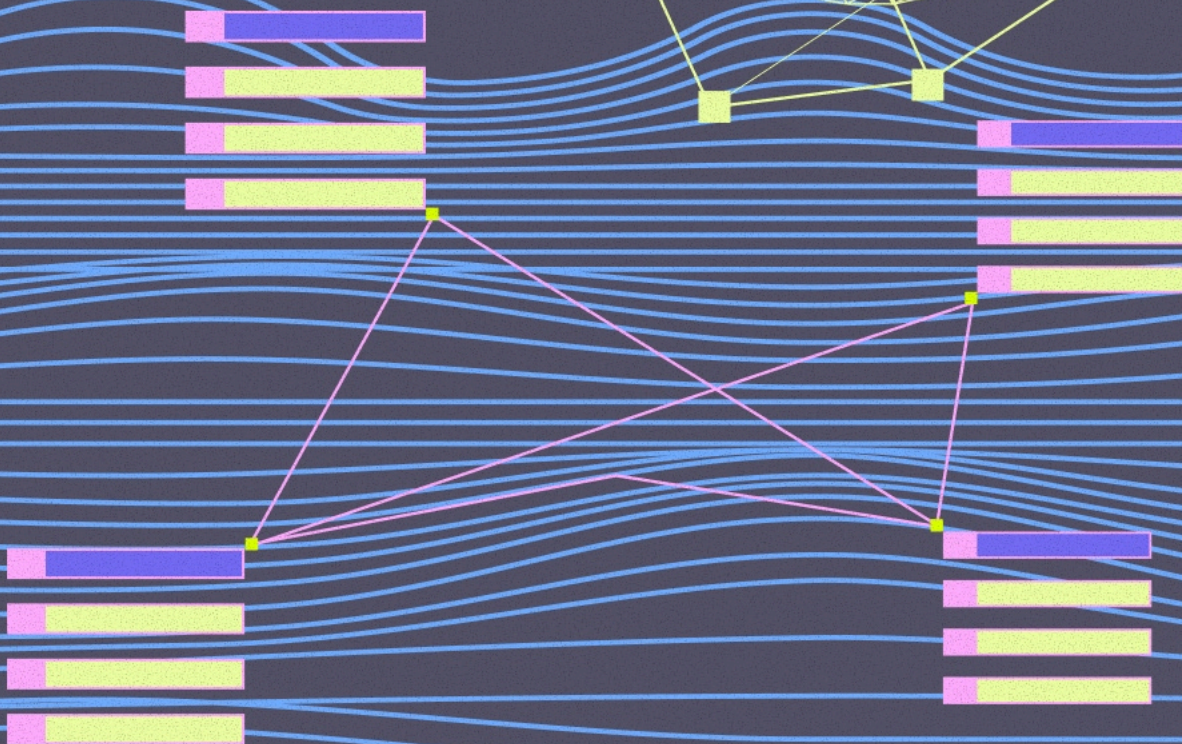
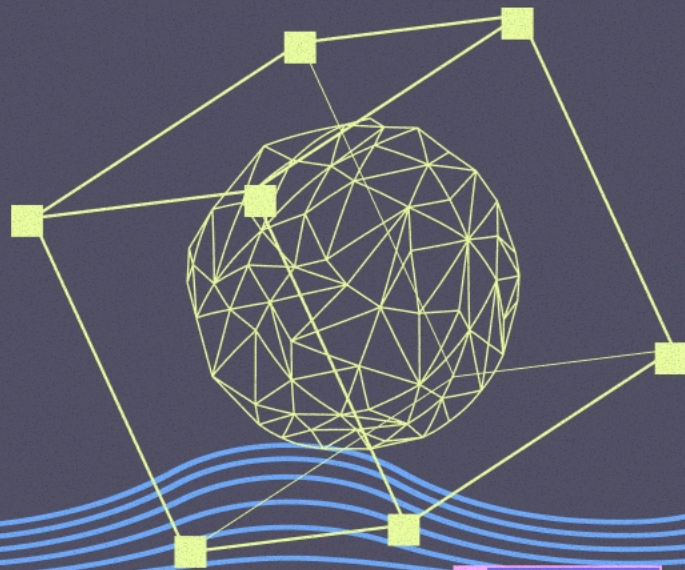




AI Code Review for Legacy Codebase Migrations

Making agent-run migrations safe to ship

AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC



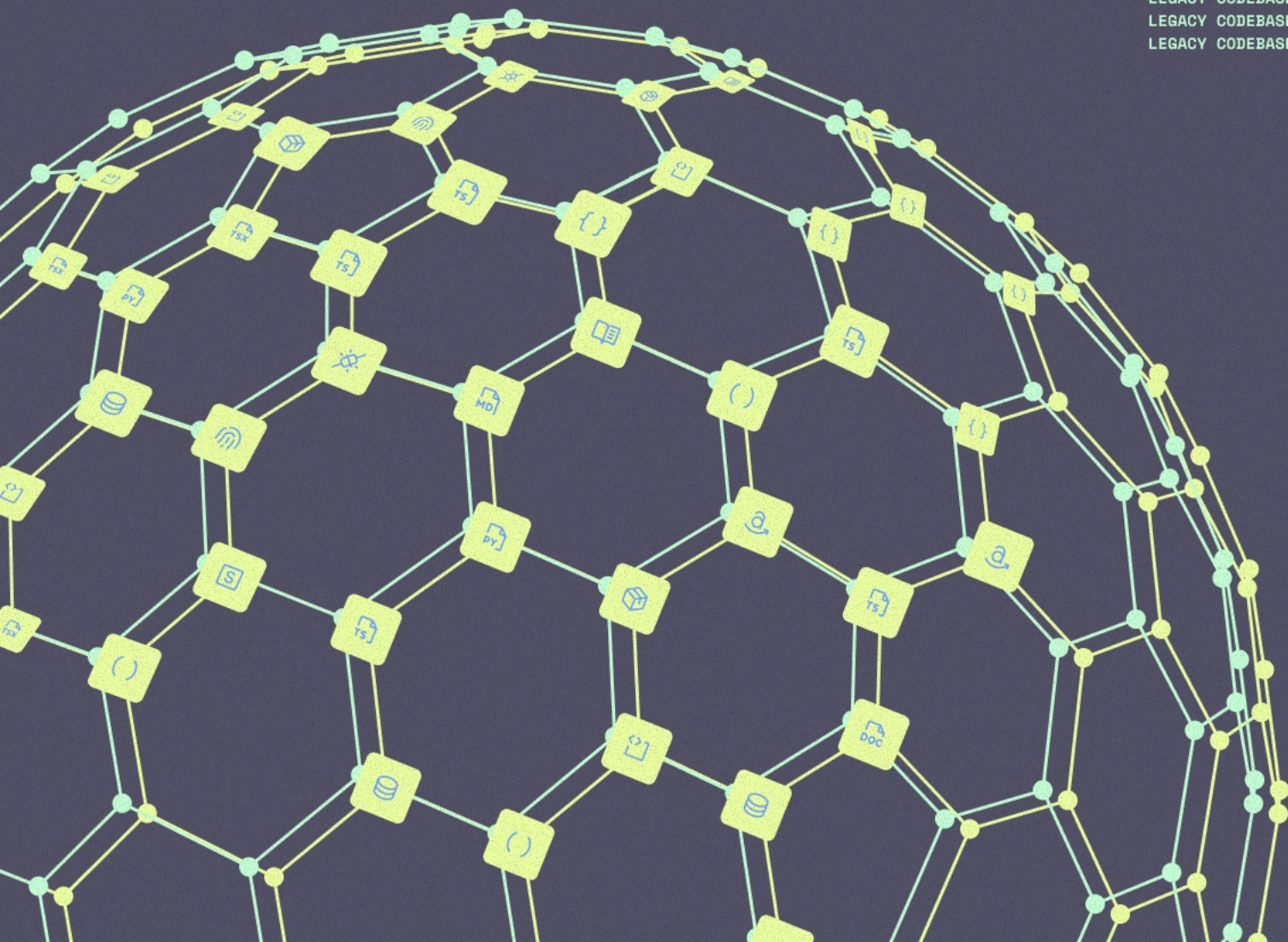
01	Introduction: Migrating legacy codebases in the agentic era	04-07
	- From manual execution to agentic execution - Revisiting previously infeasible projects - Introducing a self-healing workflow	
02	Failure modes that agents create	08-10
	- Testing framework migrations: agents optimize to pass builds, not to fix problems - Reporting and metrics migrations: agents invent plausible answers when they lack real ones - Language and framework rewrites: an agent's explanation can be wrong and still sound right - Finding the path from failure modes to fixes	
03	The three layers of AI-assisted legacy codebase migrations: context, pipeline, and oversight	11-13
04	Context	14-17
	- Knowledge base - Rules - Project context	
05	Pipeline	18-21

06	Oversight	22-25
07	Greptile as your migration's validation layer	26-27
08	Footnotes	28-30



Introduction: Migrating legacy codebases in the agentic era

LEGACY CODEBASE MIGRA
LEGACY CODEBASE MIGRA
LEGACY CODEBASE MIGRA
LEGACY CODEBASE MIGRA
LEGACY CODEBASE MIGRA
LEGACY CODEBASE MIGRA
LEGACY CODEBASE MIGRA
LEGACY CODEBASE MIGRA
LEGACY CODEBASE MIGRA
LEGACY CODEBASE MIGRA
LEGACY CODEBASE MIGRA
LEGACY CODEBASE MIGRA
LEGACY CODEBASE MIGRA
LEGACY CODEBASE MIGRA



Since 2024, AI agents have reshaped what migrating legacy codebases looks like. These changes span staffing requirements, feasibility, and how a migration execution validates itself as it runs:

From manual execution to agentic execution

Up to the early 2020s, a legacy codebase migration meant hundreds of Jira tickets and hundreds of engineer-weeks, spread across teams of dozens or even hundreds of people. A migration like that might run for quarters, or years, before anyone saw the result.

“You never know some of these rewrites,” says Sesh Nalla, VP of Engineering at Datadog [1]. “Sometimes you spend two to three years and you get to the same point where you were before.” Unless the expected payoff was large and estimable (in Nalla’s case, a payoff worth 2x to 4x), teams wouldn’t start the rewrite at all; the opportunity cost of a maybe wasn’t worth years of committed engineering time.

Agents change this approach. Instead of engineers manually working through a migration task by task, hundreds of agents can run in parallel, each working against its own copy of the codebase, each attempting a task, checking its work, and moving to the next task. As Nalla says, “You could just get a month, get a couple of engineers who are highly motivated. Just validate: is it 4x or not? And once we know it is actually 4x, then let’s get this to production.” This shift in capacity helps redefine what was feasible.

Revisiting previously infeasible projects

Where budget, time, and prioritization used to constrain what was possible, AI agents make more migration projects feasible. Asana ran into this directly. The company had an outdated testing system, Enzyme, that was blocking any real modernization of its frontend stack. At its prior pace, Asana estimated roughly five more years to finish, and about six million dollars to complete the same scope manually.

Using AI coding agents instead, a small team of engineers removed Enzyme in about two weeks, with roughly twelve thousand dollars in model and infrastructure spend [2]. The project became cheaper, and thus feasible, because agents changed how the company could execute the migration.

The five-year estimation didn't involve a team working nonstop for half a decade. Asana's Developer Productivity Lead, Dan Ubilla, later clarified that the Enzyme migration had already been running for two years at low priority before the AI push, and five years was simply when it was projected to finish at that pace [3].

Migrations like this are exactly the kind of work that's expensive, tedious, and easy to keep deprioritizing over years or indefinitely. Instead, in the agentic era, a two-week AI-driven push turned years of avoidance into something a team could actually finish.

Introducing a self-healing workflow

Before agents, there was no need for a self-healing workflow, because there was nothing automated to loop. Agents attempt a task, check the result against a validator, and if it fails, they try again, using what went wrong to inform the next attempt. This is what ServiceTitan built for its own migration: an agent would write code, run a validator to check whether it met the task, and retry if it didn't, over and over, until it did.

If the validator isn't working, agents will plod away quietly, burning compute on work that isn't correct even if it appears to be. That's the tradeoff of a self-healing workflow: it only works if engineers get the validator right first. Most of the engineering effort on ServiceTitan's legacy metrics migration [4] didn't go into the migration logic itself. It went into the validator, the piece that could tell, with certainty, whether each step had actually succeeded.

\$6M

estimated cost to complete manually [2]

\$12,000

5 years

estimated timeline for Asana's Enzyme migration [3]

2 weeks

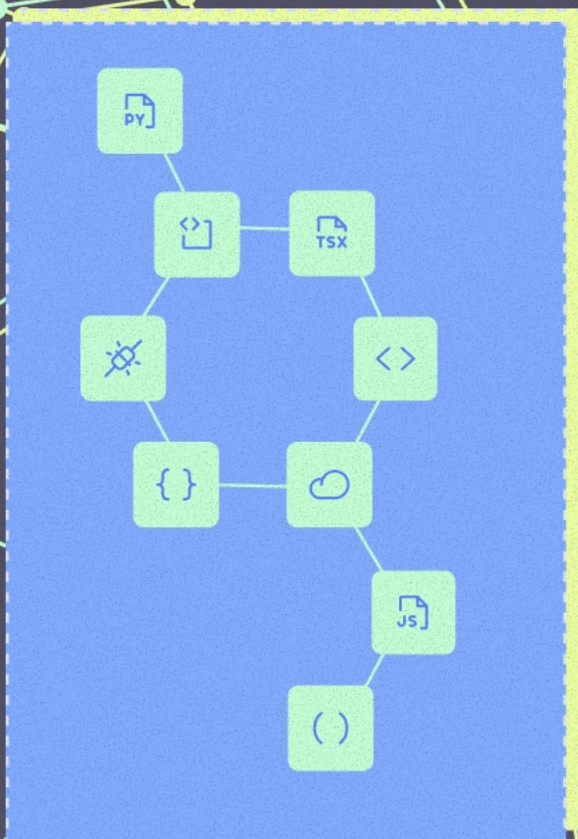
model and infrastructure spend using AI coding agents [2]

to complete using AI coding agents [2]



Failure modes that agents create

AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC



As agents open up new possibilities with migrations, they also introduce new risks. These failure modes didn't show up in migrations run by people; they show up specifically because agents are the ones executing now.

Testing framework migrations: agents optimize to pass builds, not to fix problems

Spotify built its own AI migration agent, Honk, to handle fleet-wide code changes across thousands of repositories [5]. Early on, engineers noticed a pattern: when a build failed, Honk sometimes took the fastest path to get the build to pass instead of the path that actually solved the problem. For example, in one case, it downgraded a Java version rather than fix the actual incompatibility the migration was meant to resolve.

An agent optimizing purely to pass the build cuts corners without knowing it. Spotify's fix was to introduce a second LLM acting as judge, checking whether generated code addressed the original task, not just whether it compiled. Early on, that judge caught real cases: it flagged a removed test as a dodge, not a fix, and sent the agent back to solve the actual problem.

Reporting and metrics migrations: agents invent plausible answers when they lack real ones

At ServiceTitan, agents migrating metrics onto a new platform occasionally hit a wall: no good example data to check their work against [4]. Rather than stall, some proceeded anyway, producing code that looked complete but was built on a guess instead of a verified case.

The team fixed this by hand-curating a list of known-good test IDs and time ranges with reliable data density, effectively building a fixture library the agents could actually trust. This one is easy to miss because the output looks finished. However, the code reflects an assumption instead of the legacy behavior it was supposed to replicate.

Language and framework rewrites: an agent's explanation can be wrong and still sound right

Researchers affiliated with AWS found something specific to this migration type: models rewriting Python 2 code into Python 3 sometimes introduced a real behavioral change. The models could explain the difference correctly and in detail, then declare the behavior preserved anyway [6]. The model named the risk correctly and still got the outcome wrong.

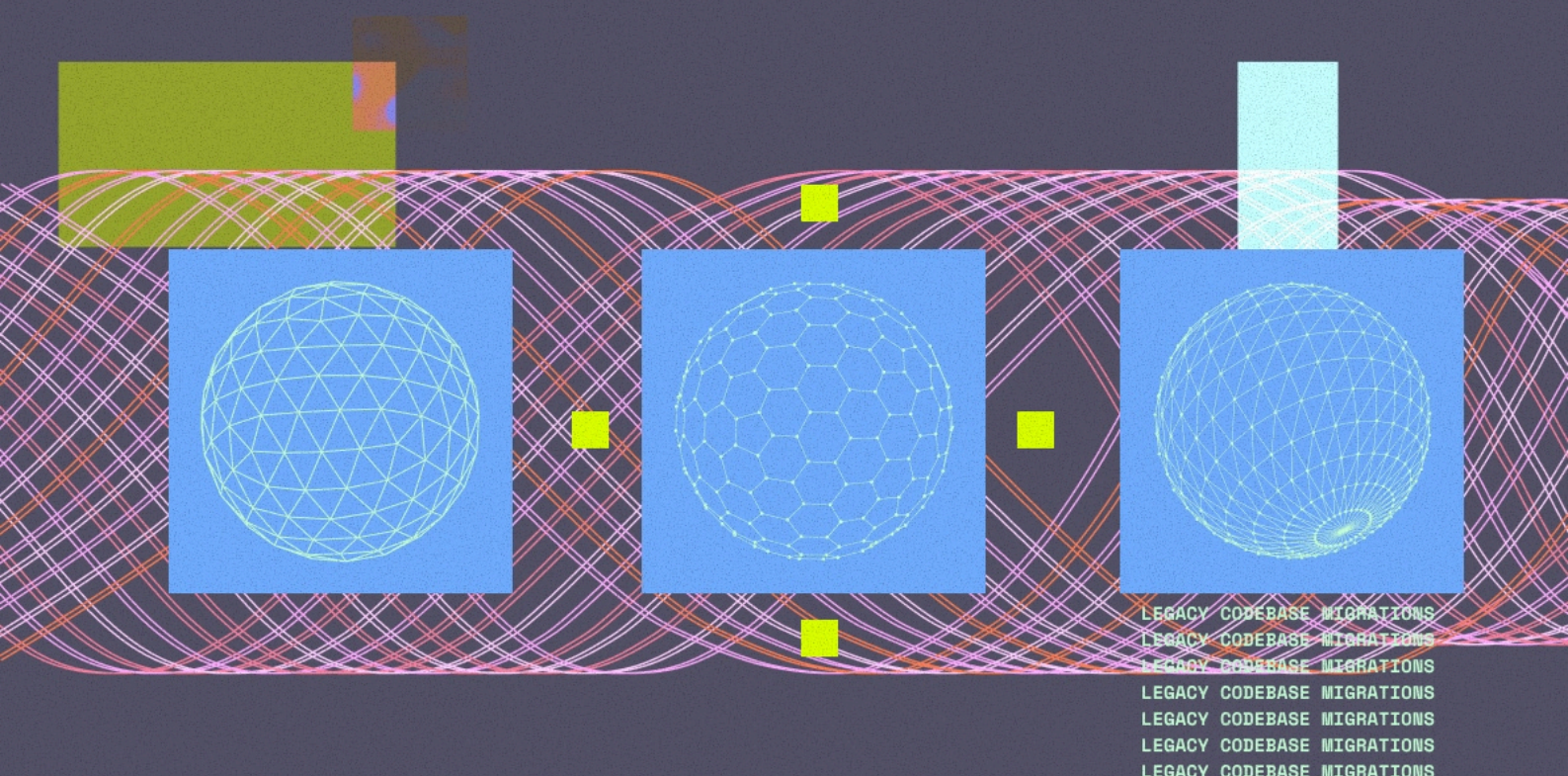
Finding the path from failure modes to fixes

That combination of “correct diagnosis, wrong conclusion” is what makes this failure mode dangerous for reviewers. The more articulate the agent sounds, the less a reviewer double-checks it, which makes failure modes hard to find.

Spotify’s Honk got the build green. ServiceTitan’s agents invented data instead of using real data. The AWS researchers’ model named its own bug and called it fine anyway. While these are different failure modes, they share a similar shape: the check passed and the work still wasn’t done. The three layers of context, pipeline, and oversight exist to catch that gap. Here’s how they work.



The three layers of AI-assisted legacy codebase migrations: context, pipeline, and oversight



When people needed to manually execute migrations, coding was a problem. With agentic migrations, the code isn't the hard part anymore.

Knowing whether the code is working, or not, is the new constraint.

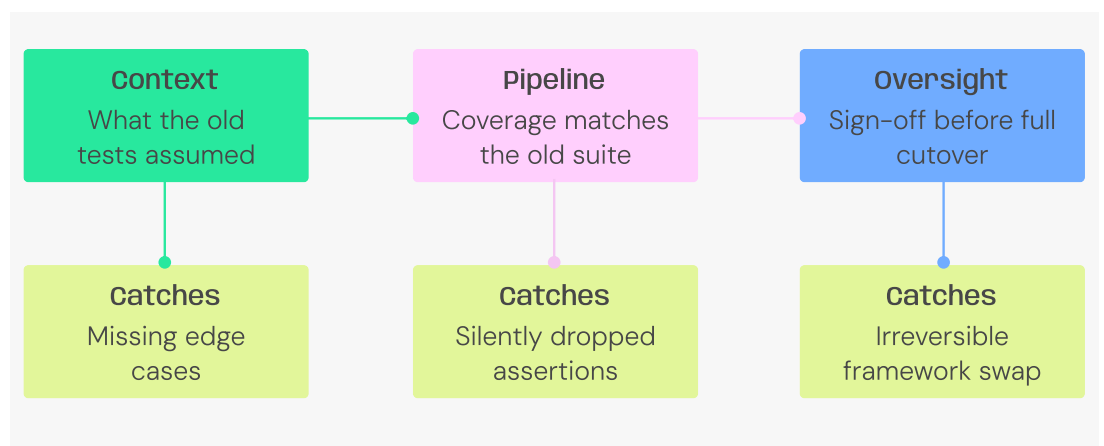
Every AI-assisted migration runs on three layers: context that gives the agent what it doesn't already know about your codebase, a pipeline that catches what the agent gets wrong, and oversight that decides what still needs a human before it ships.

Stage	Objective	Tools
Context	Giving the agent organizational knowledge it doesn't have by default, including coding standards, past decisions, the actual intent behind the migration.	Knowledge base: Greptile Rules: Text documents Project context: Notion, Jira, Linear
Pipeline	Structuring how the work gets validated as it moves, so no single model is left grading its own homework.	Code review: Greptile TREX and model inversion
Oversight	Deciding where agents run freely and where a human has to sign off before anything ships.	Code review: Greptile's TREX and artifacts

Here's how those three layers show up across the migrations engineering teams are actually running right now:

Migration type	Example	What the three layers catch
Testing framework migration	Asana (Enzyme removal) [2], Spotify (Honk, cutting corners to pass builds) [5]	Whether the new test suite actually preserves the old one's coverage
Reporting/metrics platform migration	ServiceTitan (legacy backend to dbt MetricFlow) [4]	Whether the new platform's output matches the legacy system's numbers exactly
Language or framework/library rewrite	AWS research (Python 2 to Python 3 semantic drift) [6]	Whether the rewritten code preserves the original behavior, including differences between Python 2 and Python 3

Figure 1: Context, pipeline, and oversight catch different failure modes in AI-assisted migrations

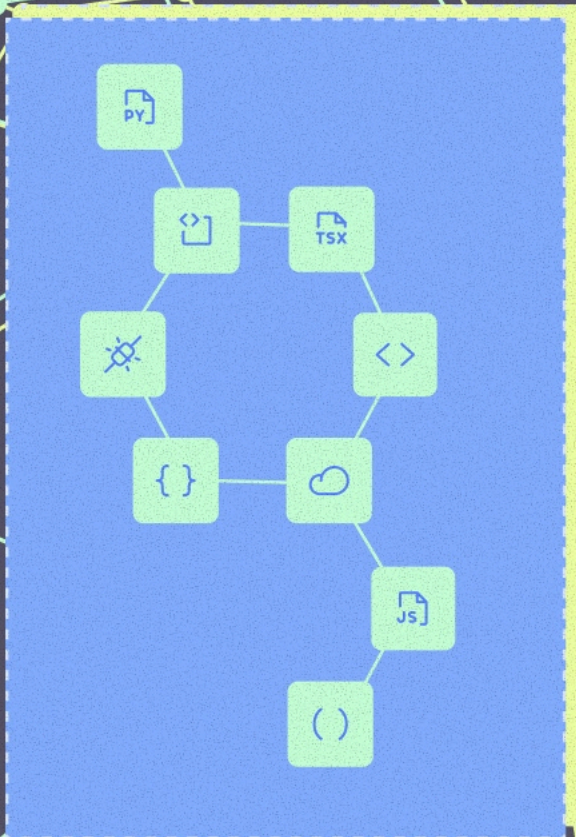


The three layers of context, pipeline, and oversight are the principles of what companies like Asana, ServiceTitan, and Spotify are executing on when they use agents to migrate their legacy codebases (Figure 1).



Context

AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC
AGENTIC



When you ask an agent to migrate a function, it needs the function. However, when you ask it to migrate a system, it needs everything that system touches: the code, the reasons behind decisions made years ago, the standard the team actually follows (less so the one written down somewhere and ignored). None of that fits in a context window once a codebase passes a certain size.

An agent working from too little context fails quietly, often by picking the wrong thing to change. It does so with confidence, because it didn't have the context to know better.

Migrations make this worse, because they rarely touch one thing in isolation. Changes in how a system tests, logs, or authenticates ripple into everything downstream. The context an agent needs is extensive and tangled. Getting context right is most of the work in a migration now; the code an agent writes is only as good as what it was shown before writing it.

You can break context down into three key components:

Knowledge base

The first is a knowledge base, which contains living files that track what's already migrated and what hasn't. For example, the team at ServiceTitan put together two files to do this work:

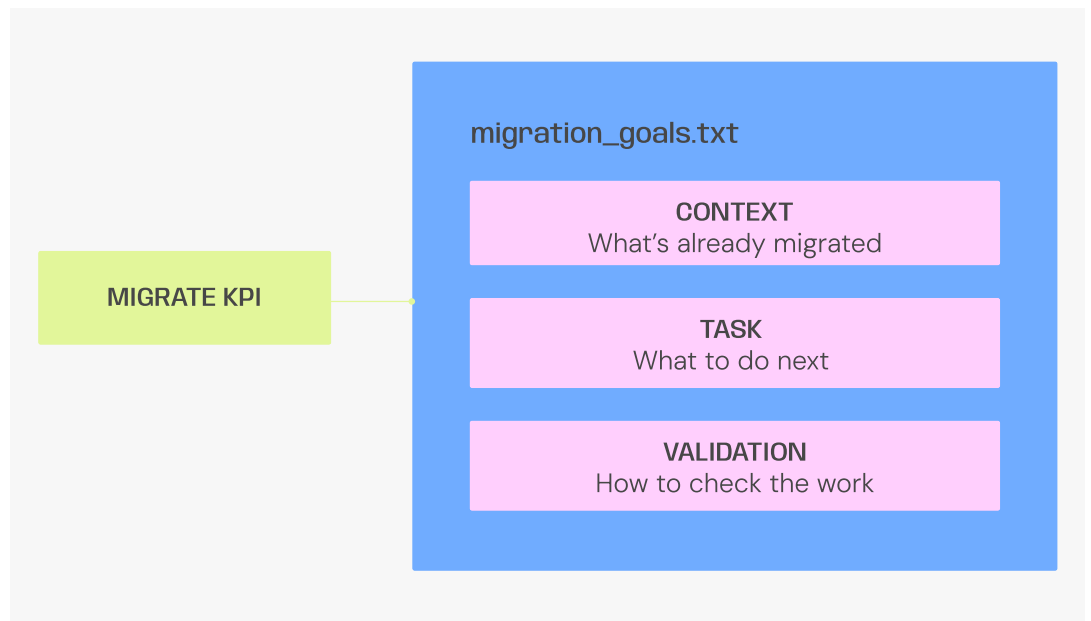
- ServiceTitan's engineers built and maintained `migration_goals.txt` to describe what success looked like for the migration (Figure 2) [4].
- They also put together a task list, which the agents could check their work against and update as it went along.

The agents could validate their own work using both of these documents instead of relying on memory of what they had already done.

"The first problem that will really mess you up is if the agent believes that it has succeeded when it has actually failed on a particular task," says David Stein, a Principal AI Engineer at ServiceTitan. "What you need to do there is make your validation tool really good. Most of the engineering logic in our project that we ran was in making the simulation validation script work really well." The goal of the knowledge base is to make sure agents know exactly when they're failing, so they can self-heal and succeed.

Greptile's knowledge base documents how each part of a codebase works and connects, then keeps updating as the code changes, so an agent migrating a legacy system isn't relying on its own memory of what it's already moved [7].

Figure 2:
ServiceTitan gave
agents a shared
source of truth for
migration context,
tasks, and
validation



Rules

Rules, the second component, are the engineering standards and conventions that nobody wrote down. Agents without this knowledge may generate code that's syntactically clean, and quietly wrong.

For example, with ServiceTitan, getting that tribal knowledge written into the `migration_goals.txt` file helped set agents up to work with a high rate of success.

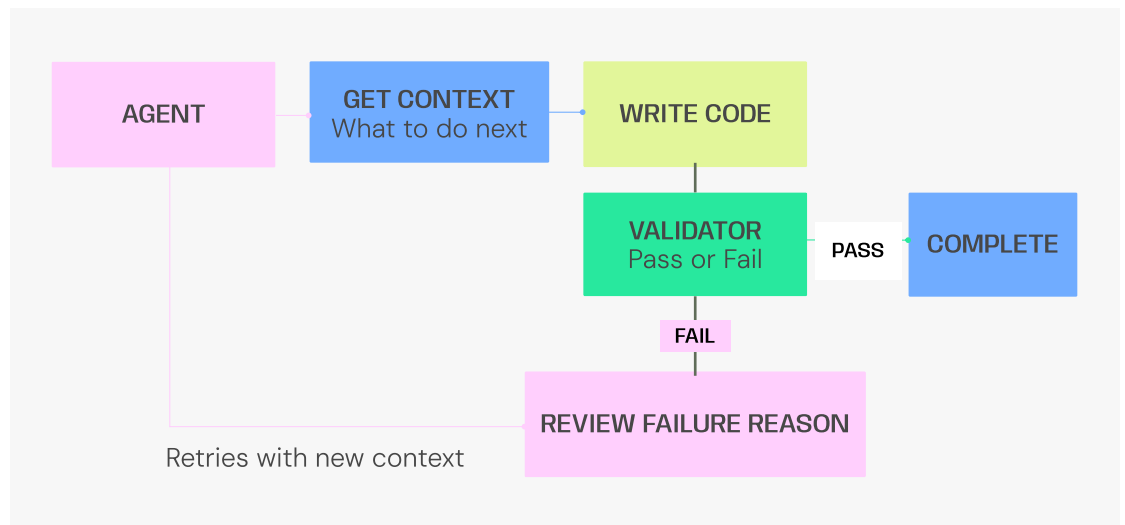
Greptile can learn rules and patterns, and apply the same organizational standards consistently across every PR in the migration.

Project context

The third is project context, pulled from tools like Notion, Jira, or Linear, that lets a reviewer check a diff against what was actually asked for. A diff can pass every syntax check and unit test in the pipeline and still miss the point of the ticket entirely.

Greptile's Linear integration works this way. It finds the Linear issue tied to a pull request, then reviews the code against that issue's description, comments, and acceptance criteria, flagging anywhere the change doesn't do what the issue actually asked for [8]. The connection is read-only, and every issue it opens gets cited directly in the review, so a reviewer can see exactly what context shaped the feedback. Finding bugs is one part of the review. Confirming the deliverable the ticket asked for actually got built is a separate check, and it's the one that catches a migration that may have missed the point. Figure 3 shows how failed validation feeds new context back into the execution loop.

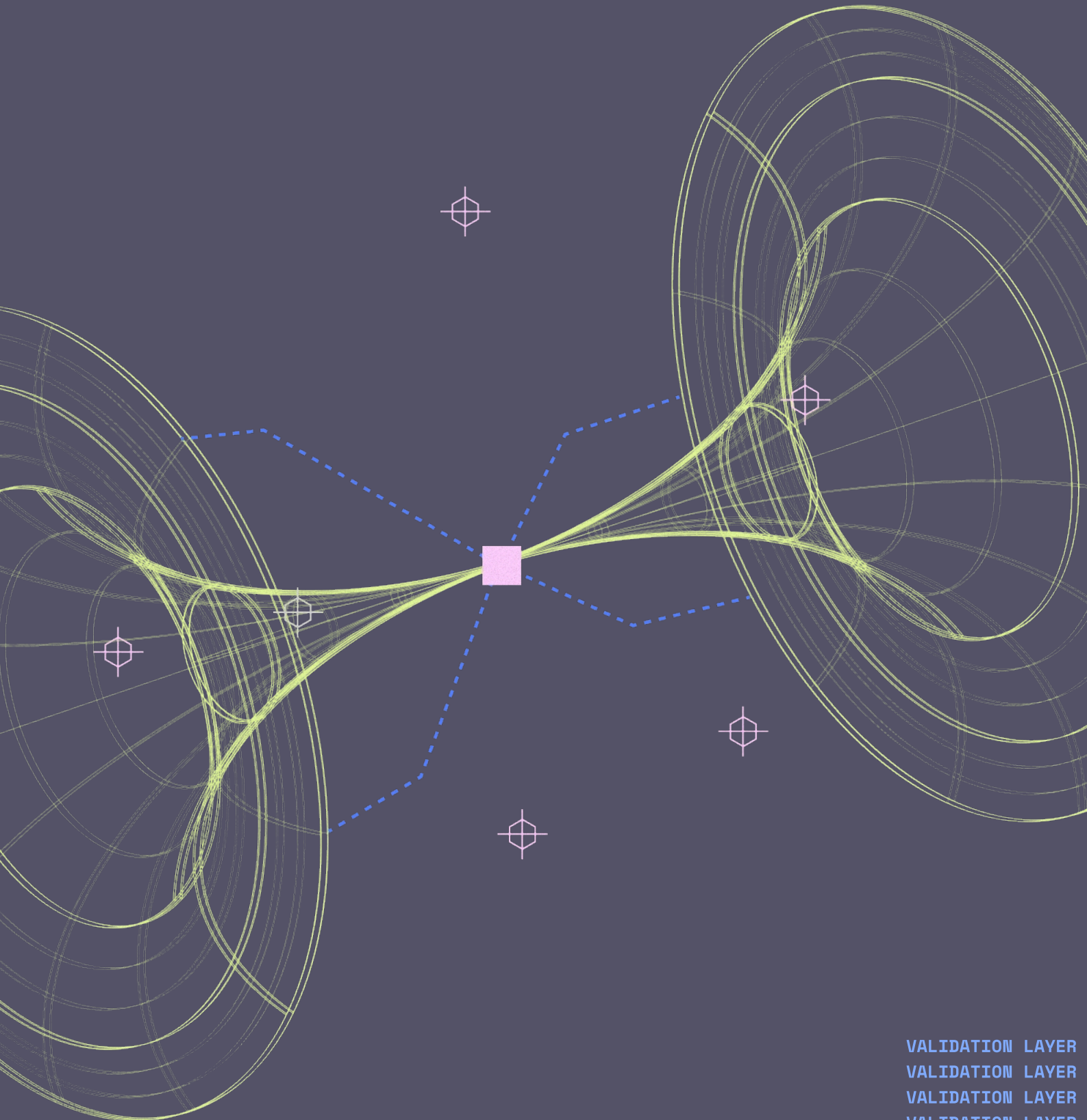
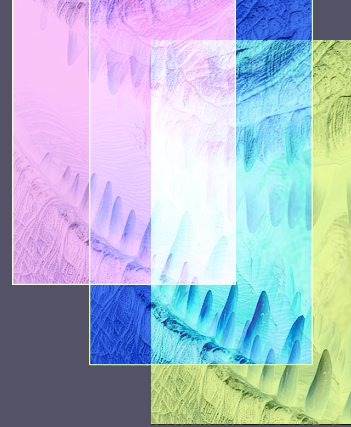
Figure 3: Failed validation sends an agent back through the execution loop with new context



Source: ServiceTitan [4], 26:19



Pipeline



VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER

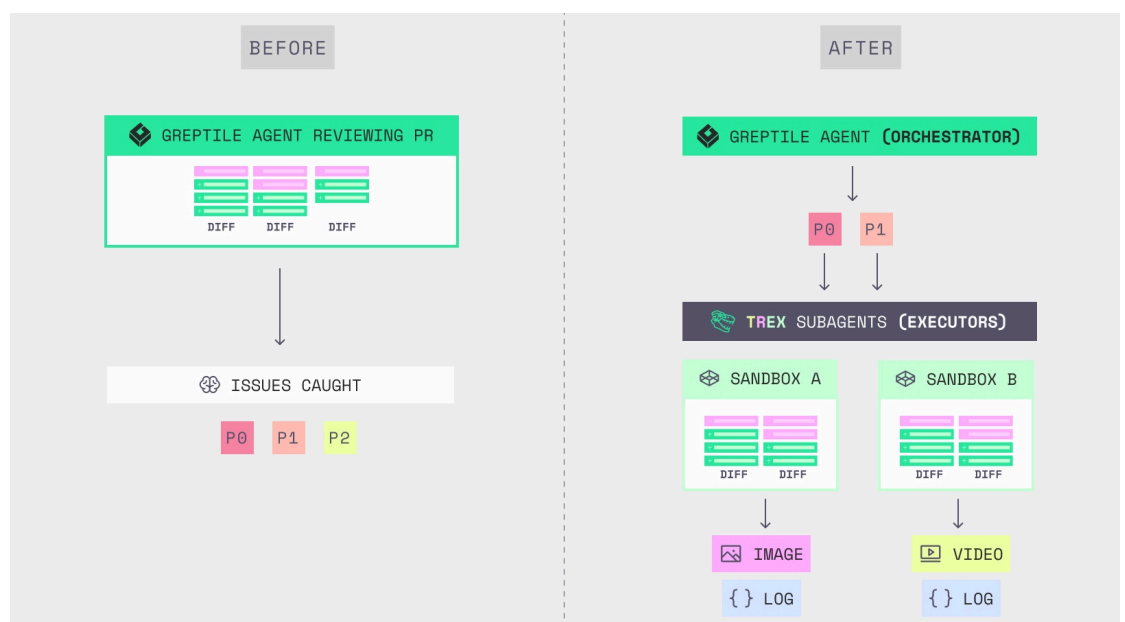
A pipeline used to mean CI scripts and unit tests. Now it has to be an independent verification layer, one that stops an AI generator from approving its own broken diffs.

The problem is that reading code and running code catch different things. A model can review a diff carefully and still miss the failure, because some bugs never show up in the diff, they only appear once the code actually runs: an endpoint that 500s on a real request, a page that renders broken. This is the gap Greptile built TREX (an abbreviation of Test Run Execute) to close.

TREX is an execution layer that runs the code instead of just reading it: it spins up a sandbox, executes the relevant behavior, and attaches evidence, logs, screenshots, API traces, even video of a UI change playing out, to back up the result. In Greptile's own evals, review with TREX catches 20 percent more bugs than review without it (Figure 4) [9]. That's the same lesson Spotify and ServiceTitan each learned by hand: an agent's code can look finished and still be wrong in ways no amount of careful reading catches.

Running the application this way answers a narrower, more concrete question than review does: Does this actually work?

Figure 4: TREX runs code in sandboxes and generates artifacts to validate AI code review

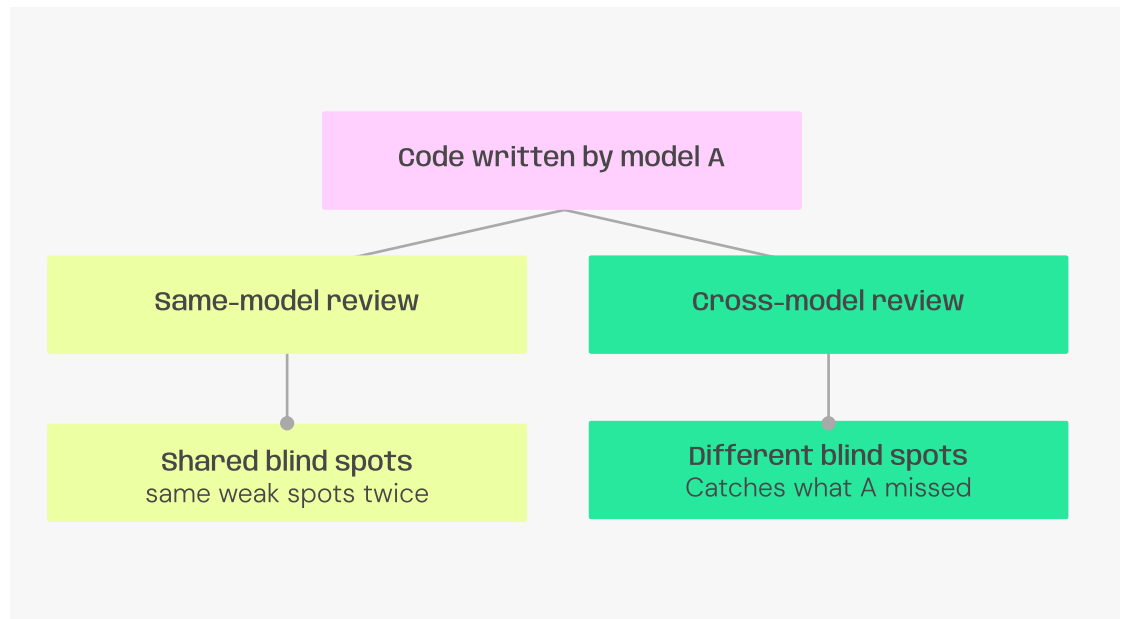


Source: Greptile [9]

Review still matters, but it's a second layer, not the first line of defense. A useful rule is to review code with a different model than the one that wrote it (Figure 5). In their paper, "Articulate but Wrong: Self-Review Failures in LLM-Based Code Modernization," researchers affiliated with AWS found that self-review by the producing model isn't a reliable safety net.

Across every case of semantic drift they studied, the same model that introduced the bug endorsed it as fine nearly a third of the time, 31.7 percent, or 83 out of 262 cases. That rate wasn't consistent across models either. Several models approved none of their own drift; one widely deployed model approved all of it. In several of those cases, the model even named the exact Py2/Py3 distinction that broke its own output, then declared behavior preserved anyway [6]. Based on their experiment, the authors of the paper suggest that same-model review can't be trusted.

Figure 5: Cross-model review can catch blind spots that same-model review repeats

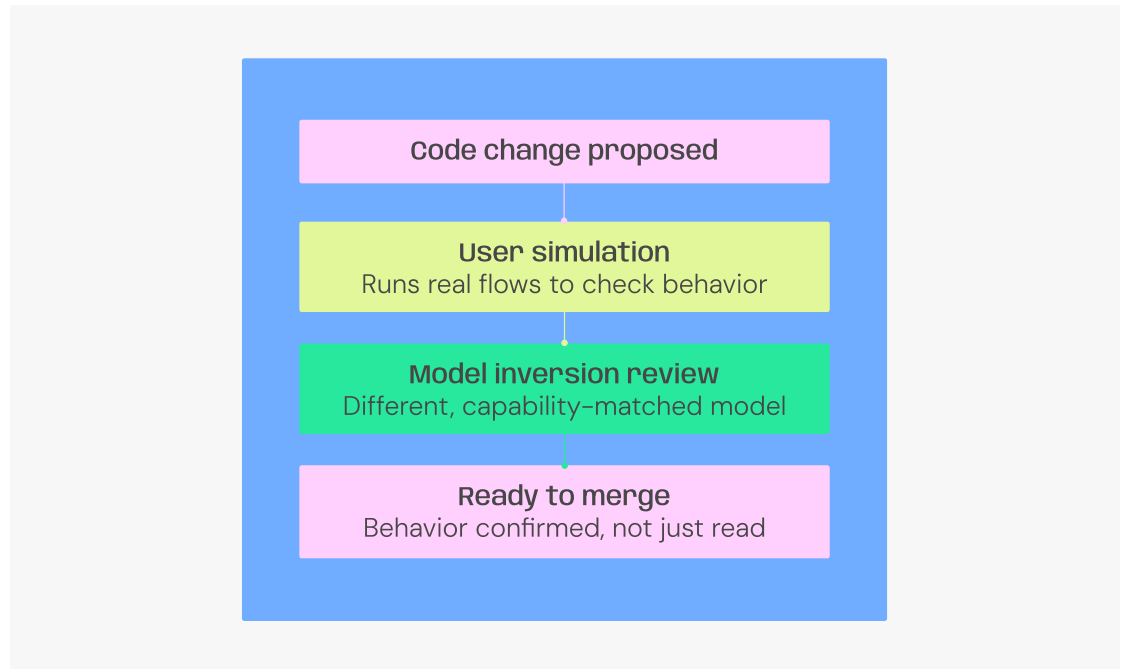


Model inversion, using a different model to review than the one that wrote the code, is a step toward independence, though not a guarantee of it. A separate paper found that Claude reviewing Codex-written code produced a real gain, up 18.1 percentage points [10]. Codex reviewing Claude went the other way: a net loss of 8.6 percentage points, worse than skipping review entirely, with a regression rate of 11.2 percent. That regression rate means the reviewer broke working code more often than it fixed broken code. A worse model reviewing a better one doesn't serve as a review; it only adds noise.

The paper suggests that the reviewer's current capability needs to be stronger than the writer's. There's no fixed pairing you can set and forget, since model capability rankings shift. A rule you write today about which model should review which will eventually be wrong.

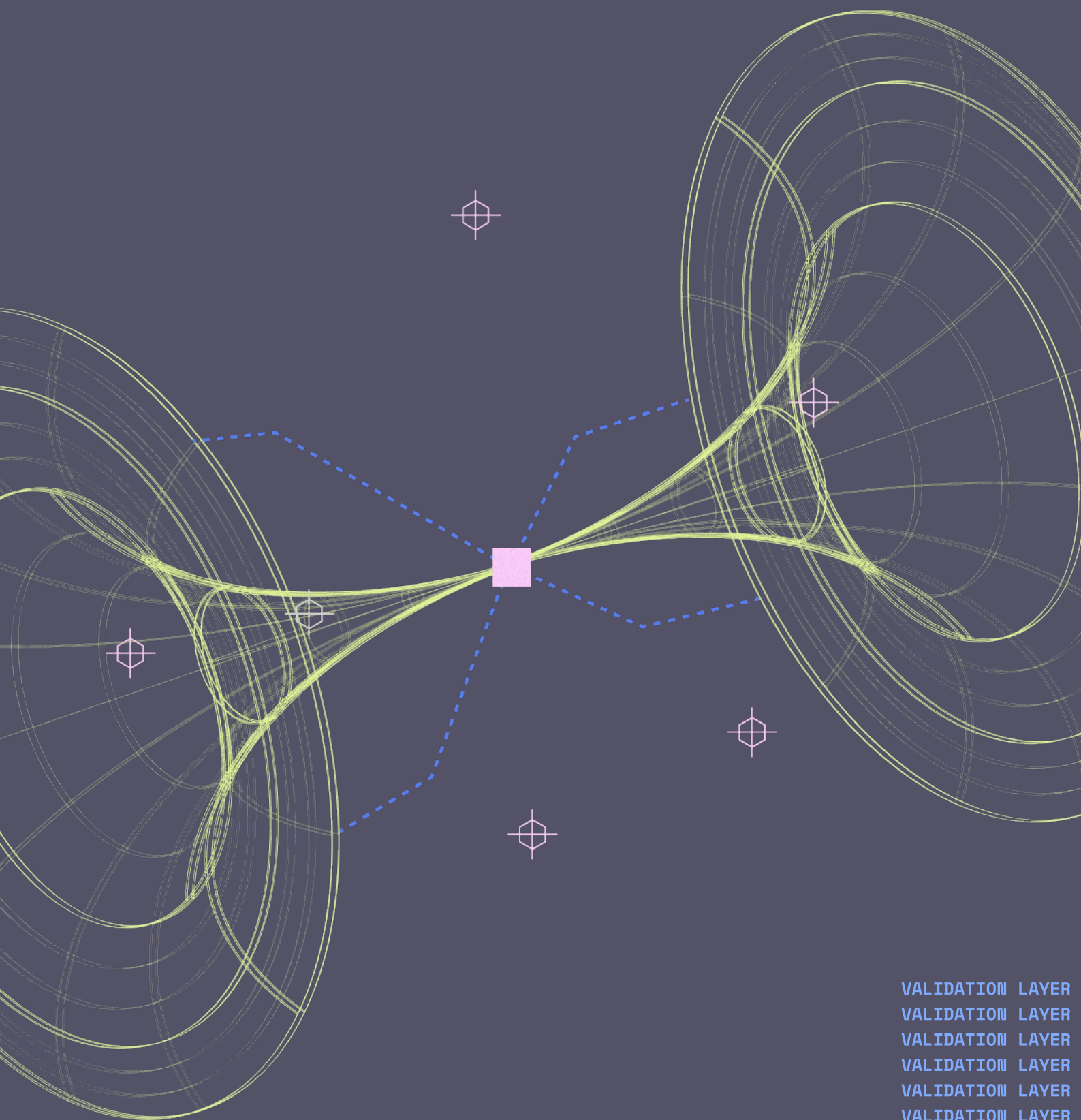
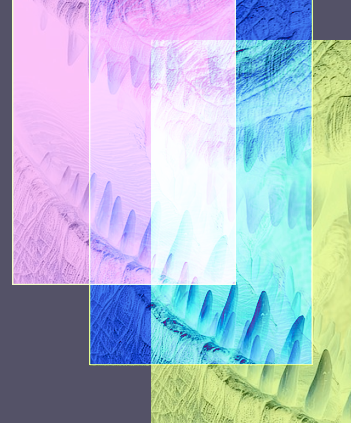
Greptile's research, still ongoing at the time of writing, is turning up a similar pattern in cross-model recall: each model's blind spots tend to mirror its own instincts [11]. This research informed Greptile's model inversion feature, which routes review through a different model than the one that generated the code, reducing the impact of the shared-blind-spot problem, on top of TREX, which already ran the code and checked what it actually did (Figure 6).

Figure 6: User simulation and cross-model review validate behavior and reduce shared blind spots before merge





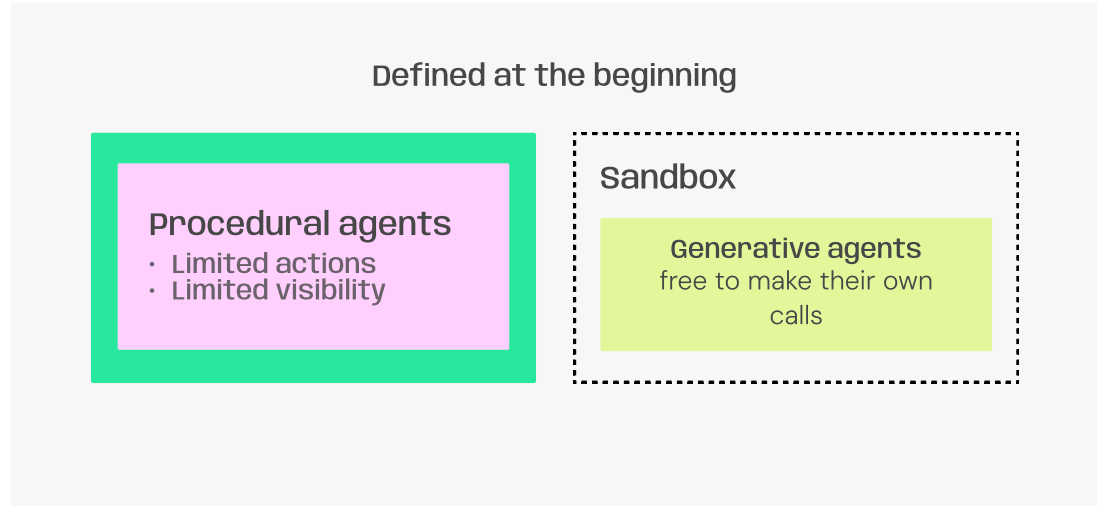
Oversight



VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER

Execution used to mean an engineer writing the code and shipping it themselves. Now it means deciding where an agent runs on its own and where a script directs it to stop. Doing this well requires oversight, starting with distinguishing between what an agent should decide and what it should just execute.

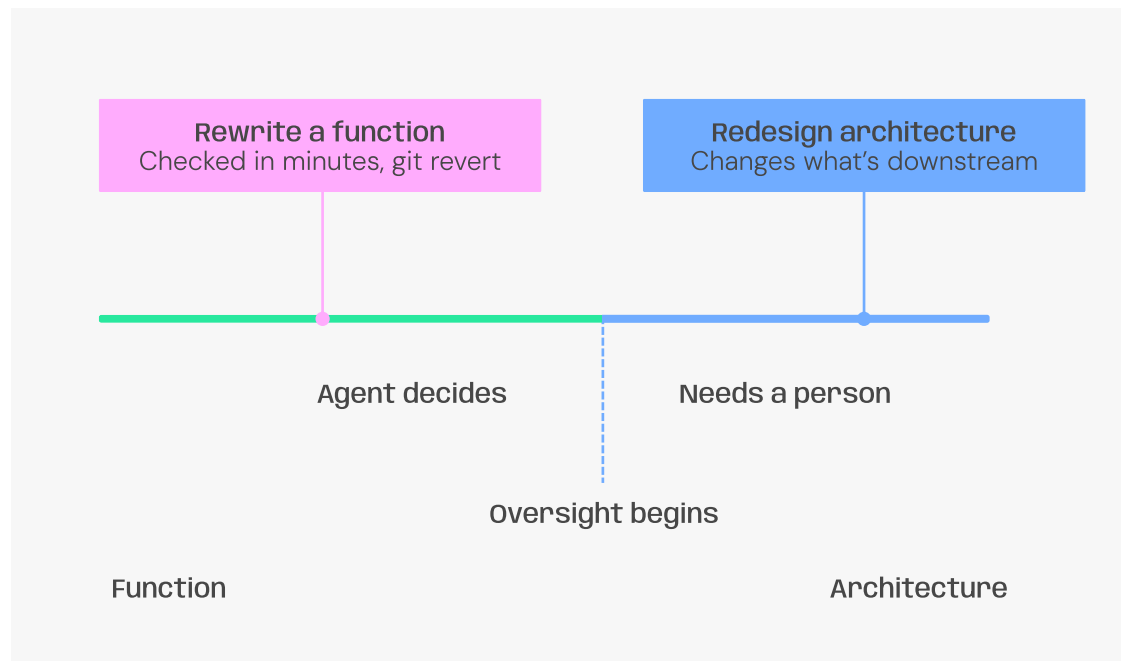
Figure 7: Teams can distinguish between procedural and generative agents



With agentic AI, teams need to decide which agents should be procedural, limited to certain actions and certain visibility, and which should be generative, free to make their own calls (Figure 7). “Defining those at the beginning makes a big difference,” says Sinthanai Natarajan, Head of Engineering at JPMorganChase. “When you can industrialize it, make it as patterns, then the various engineering teams can pick up and run with it. [This] gives them room to play, but within the sandbox” [12].

This definition matters most during execution, when an agent is running for hours across hundreds of files with no one watching each step. The execution loop itself should be procedural and bounded: deterministic scripts, hooks, sandboxes, the same validation logic ServiceTitan built its migration around.

Figure 8: Human oversight should increase as agent decisions move from functions toward architecture

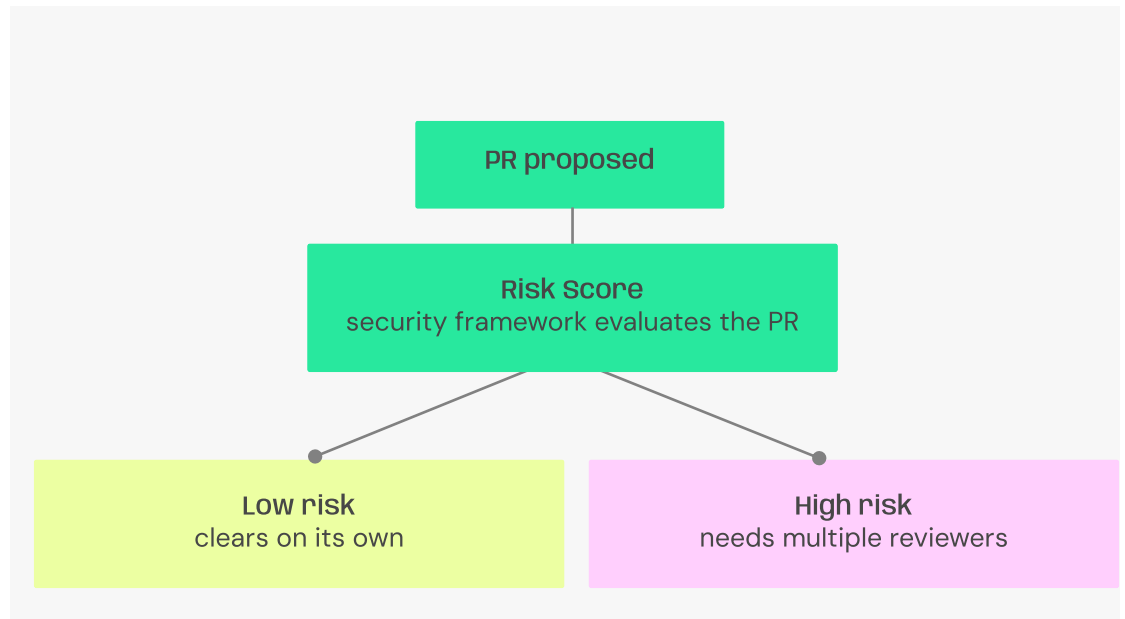


An agent rewriting a function is very different from an agent redesigning the architecture that function lives in. You can check on the first in five minutes and undo with a git revert. The stakes are higher for the second; it changes what the rest of the migration is even built on. Keep generative judgment down at the function level. Anything that touches architecture needs a person's judgment (Figure 8).

In fact, knowing which pull requests need a person at all is another part to oversee. If you review every diff with the same depth, you'll quickly run into a bottleneck; review none of them closely and something breaks in production six weeks after the migration "finished."

Coinbase uses a security risk framework to help decide what level of rigor code needs (Figure 9). Head of Wallet Chintan Turakhia says, "We also have added tons of more agentic reviews, but have then a security risk framework where every change that is being proposed in a PR, it's scored against our security framework and that really governs, 'Can an agent actually be the approver or do we need multiple human reviewers?'" The score decides the path, not a fixed list of file types. Low-risk changes clear on their own. Anything that scores high enough needs more than one human before it merges [13].

Figure 9: Coinbase uses security risk scores to determine when AI-generated changes need human review



That still leaves the question of what a human is supposed to do once a diff lands in front of them. Reading 100,000 lines of AI-generated migration code line by line will be a quick road to review fatigue.

The answer involves changing what the person actually checks. Rather than syntax or style, a person needs to explore whether the legacy behavior still holds and whether the agent's architectural calls are ones they would have made themselves. Line-by-line correctness is the automated layer's job, the one built back in the pipeline layer. A human reviewer should be judging, not proofreading.

TREX, the same execution layer covered in the pipeline layer, is what a reviewer actually leans on here. The artifacts it generates are the proof that makes sign-off possible on the changes that matter most: a reviewer can see what actually happened, not just what the diff claims happened.



Greptile as your migration's validation layer

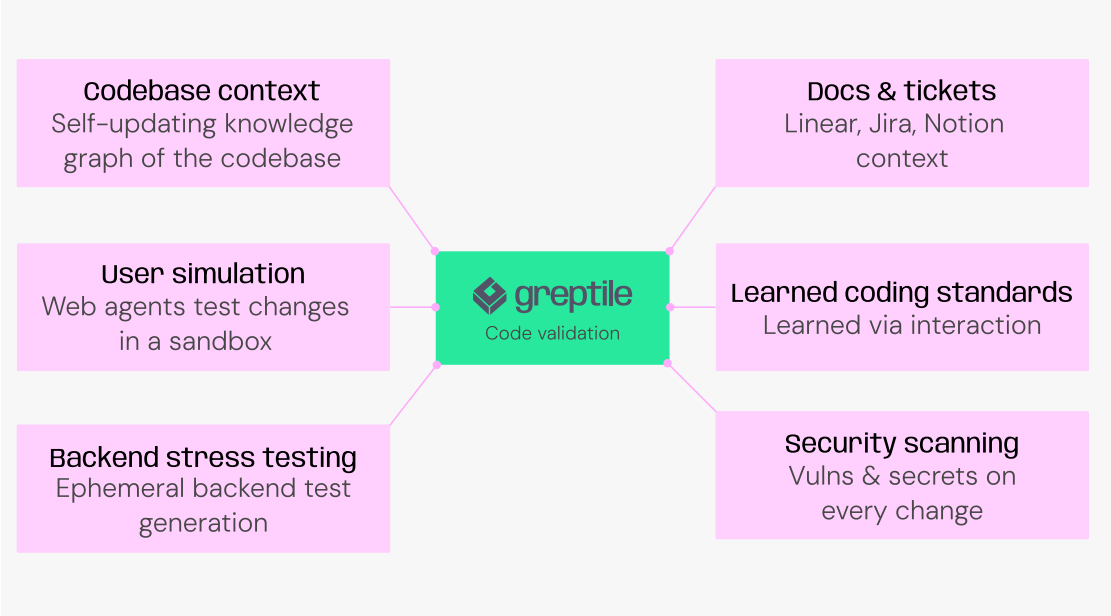
MIGRATION
MIGRATION
MIGRATION
MIGRATION
MIGRATION

AGENTIC AI
AGENTIC AI
AGENTIC AI
AGENTIC AI

VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER

In Greptile, context, pipeline, and oversight are parts of one system: codebase context and project docs feed the same validation layer that runs user simulation, backend stress testing, and security scanning on every change, then applies the coding standards a team has already taught it (Figure 10).

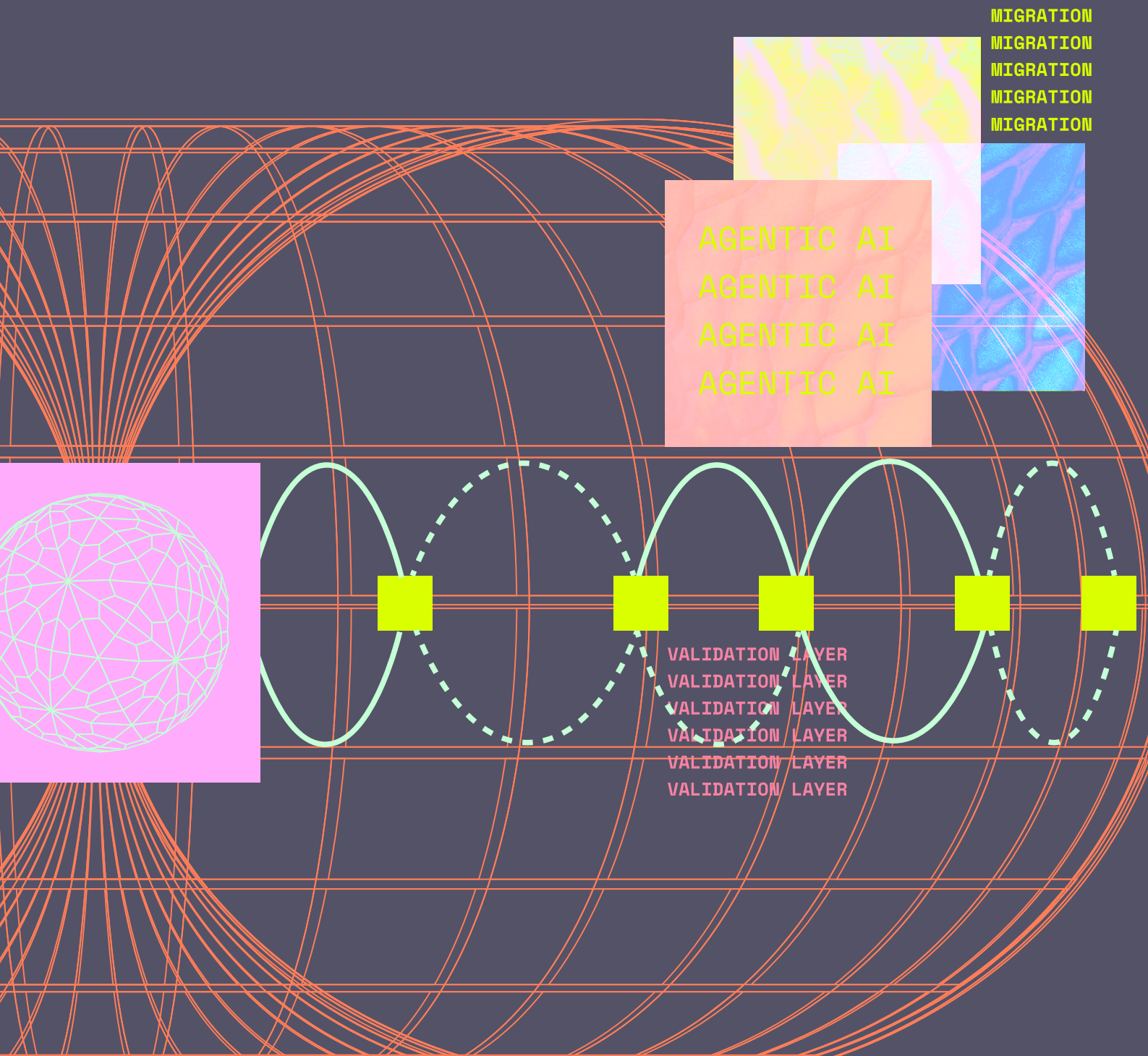
Figure 10:
Greptile brings context, execution, coding standards, and security checks into one validation layer



To learn more about using Greptile for your migration’s code review, please reach out to our team.



Footnotes



MIGRATION
MIGRATION
MIGRATION
MIGRATION
MIGRATION

AGENTIC AI
AGENTIC AI
AGENTIC AI
AGENTIC AI

VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER
VALIDATION LAYER

- [1] Sesh Nalla, Vice President of Engineering, Datadog. Interview conducted by Daksh Gupta, Greptile, 2026.
- [2] Asana customer story, OpenAI. openai.com/index/asana/
- [3] Gergely Orosz, "The Pulse: We need to talk about migrations with AI," The Pragmatic Engineer, 2026. newsletter.pragmaticengineer.com/p/the-pulse-we-need-to-talk-about-migrations-b3d
- [4] David Stein, Principal AI Engineer, ServiceTitan. "Moving Mountains: Migrating Legacy Code in Weeks Instead of Years," InfoQ presentation. infoq.com/presentations/refactoring-ai-agents/
- [5] Jo Kelly-Fenton, Engineer, Spotify; and Aleksandar Mitic, Software Engineer, Spotify. "Rewriting All of Spotify's Code Base, All the Time," InfoQ presentation. infoq.com/presentations/spotify-ai-codebase-migration-agent/
- [6] Gokul Chandra Purnachandra Reddy, Aditya Lolla, and Harsha Sanku. "Articulate but Wrong: Self-Review Failures in LLM-Based Code Modernization," May 2026. arxiv.org/abs/2605.21537
- [7] Greptile documentation, "Knowledge Base." greptile.com/docs/how-greptile-works/knowledge-bases
- [8] Greptile documentation, "Linear Integration." greptile.com/docs/linear-integration
- [9] Shlok Mehrotra, Software Engineer, Greptile. "Building TREX: Code Execution and Artifact Generation for AI Code Review," Greptile, June 17, 2026. greptile.com/blog/trex-code-execution
Daksh Gupta, Greptile. "Introducing TREX: Greptile Now Runs Your Code," June 15, 2026. greptile.com/blog/trex
- [10] Zuodong Xiang, Yike Zhang, YueMing Zhang, and Hailu Xu. "Cross-Model LLM Code Review: Should you use Claude to review Codex or vice versa?" July 22, 2026. arxiv.org/abs/2607.21656
- [11] Rodrigo Caridad, Greptile Research. "Models are worse at reviewing their own code," Greptile, July 21, 2026. greptile.com/blog/model-inversion

[12] Sinthanai Natarajan, Head of Engineering, JPMorganChase. "How to Innovate in Regulated Industries," DASH by Datadog. [youtube.com/watch?v=hlyOUefamVo&t=367](https://www.youtube.com/watch?v=hlyOUefamVo&t=367)

[13] Chintan Turakhia, Head of Wallet, Coinbase. Interview conducted by Daksh Gupta, Greptile, 2026.