AI in a production workflow

One piece of search work, from a customer complaint to production

Luan Nguyen · Lead Engineer, Envato

Shared with the VAIT community after the webinar, October 2026. The speaker notes are the talk; press S to read them, or use the PDF, which prints them after each slide.

Speaker notes

Hi everyone, thanks for having me, and thanks to the VAIT organisers for putting this on. I'm here to share how we at Envato have brought AI into our production workflow: what we actually run, what it caught, and what has not worked yet. Not a demo. One real piece of work, start to finish, and then the setup underneath it. I'll leave time for questions at the end. Two questions came in on the RSVP form and I answer both inside the talk rather than at the end. Q1: how to run a cost-effective dev environment while keeping model results consistent with production. Q2: how to avoid tech debt when AI takes away the struggle of working out how. The slides that carry an answer have a small marker in the top-right corner.

WHO'S TALKING

Luan Nguyen

Lead Engineer, Envato · App Search & Content

13 years of full-stack software engineering

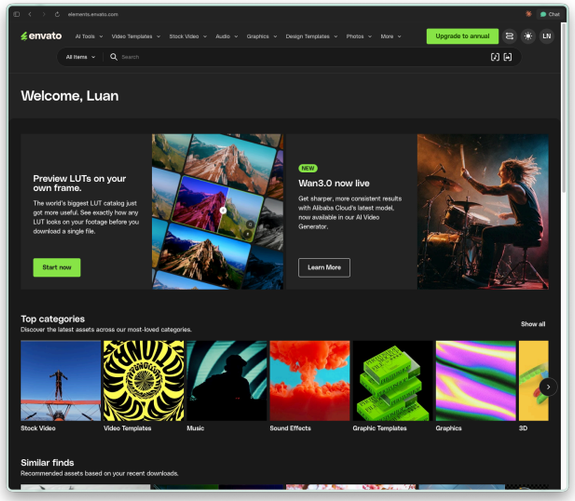
Vice President, VAIT

Speaker notes

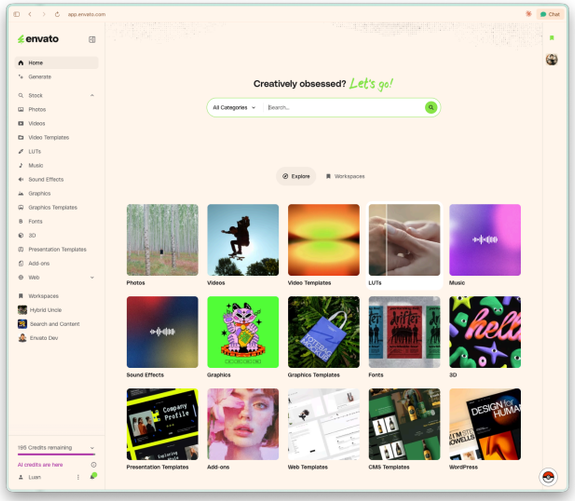
Quick introductions, three of them, then we start. Me first. I'm a lead engineer at Envato on the App Search and Content team, and my focus is search: relevance, ranking, the vector search behind "find me footage of a beach at sunset". I've been doing full-stack software engineering for thirteen years, across marketplaces and search platforms. Outside work I'm Vice President of VAIT.

WHERE THE WORK HAPPENS

Envato, and the new Envato App



Envato Elements, today

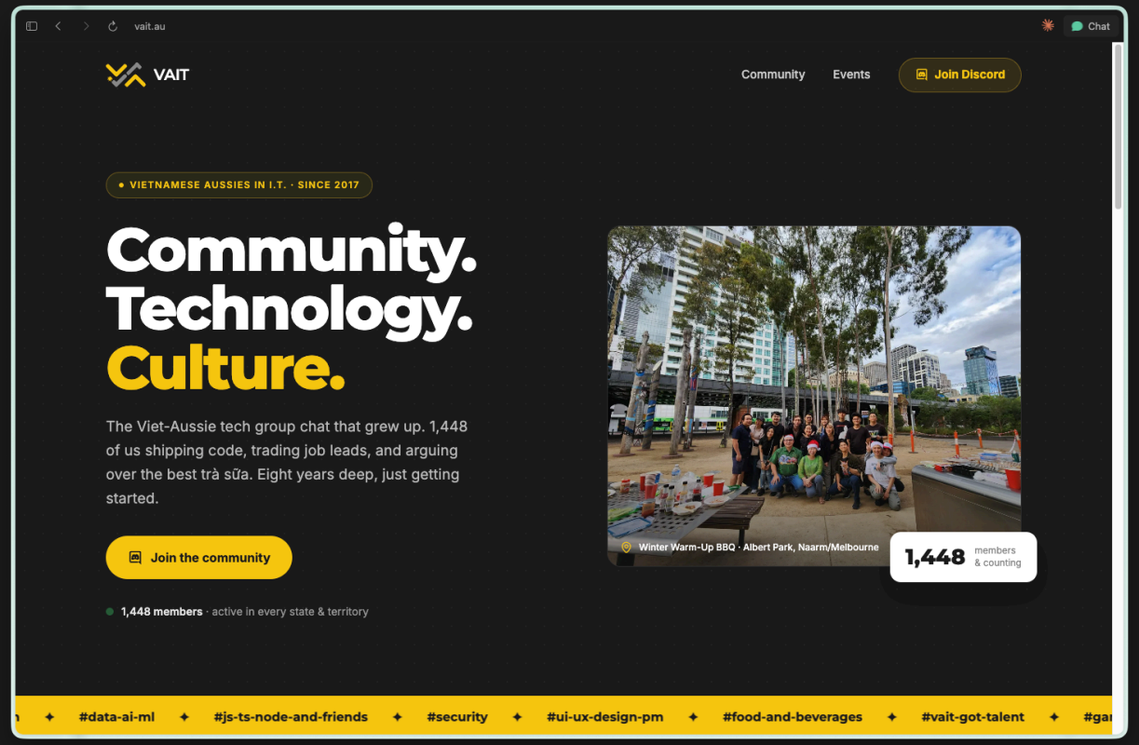


The new Envato App

Speaker notes

Envato is an Australian company, started in Melbourne in 2006. We sell creative assets by subscription: video templates, stock footage, music, graphics, fonts, photos, millions of items from a global community of authors, plus a growing set of AI generators for images, video, music and voice. The codebase in this talk is the new Envato App: a fresh product the company is building to replace the current experience, with a Rails API and a React front end in one monorepo, and a team of about sixty engineers committing to it. It is under SOX controls because it takes money. I'll explain what that means in a minute, because it shapes everything in this talk. When I say "our team" or "our pipeline", that is the one I mean.

WHO THIS IS FOR



vait.au

Speaker notes

And VAIT. The Viet-Aussie tech group chat that grew up: a community for Vietnamese Australians working in IT, going since 2017, 1,448 members and counting, active in every state and territory, built on three things, community, technology and culture. We run mock interviews, CV review weeks, meetups in Victoria and online, and sessions like this one where someone opens up how their team actually works. If you're new here, the website is vait.au, and the organisers will drop the links in the chat. This talk is made for this room: engineers early to mid career who are already using AI tools and want to see what it looks like inside a production team, with the rough edges left in.

Tonight

Two numbers

Part 1 · One piece of work, six steps

Part 2 · What's underneath

What the team said, and what is not done

Start Monday

Questions

Speaker notes

Here's the shape of the next 25 minutes. Two numbers to frame it. Then part one: one real piece of search work, followed through six steps from a customer complaint to production, and at each step where AI sat and where a person decided. Part two: the setup underneath that makes it work, the rulebook, the agents, the reviewer, the evidence. Then what the team actually said about it over seven months, and what is honestly not done yet. Six things you can start on Monday. Then your questions. But first, one piece of context that explains why our rules are as strict as they are.

BUT FIRST

What is SOX?

Who changed the code that touches money, who approved it, and that they were two different people

Speaker notes

SOX is the Sarbanes-Oxley Act, a US law from 2002, written after the Enron and WorldCom accounting scandals. The part that reaches engineers is section 404: a listed company has to prove it has internal controls over financial reporting, and auditors test those controls every year. It applies to us because our parent company is listed in the US, and because the app takes money. Any code path that can change what a customer is charged, what they are entitled to, or whether they receive what they paid for, we call it pricing, quantity and delivery, is inside the control boundary. In practice it comes down to three things an auditor will ask about any change on those paths. Who made it. Who reviewed and approved it. And were those two different people, with the evidence to show it. Segregation of duties, an approval trail, and nothing merged without review. That is why this is not a talk about convenience. When we ask "should the AI reviewer be allowed to approve", it is a control question. The approver has to be an identity the auditors can trace, acting on criteria they can read, with evidence they can check. Hold that thought for the zero on the next slides, and for why we took approval rights away from the bot in April. It also means the honesty later in the talk is not optional. If a control is not working yet, the audit trail has to say so. Right. Two numbers.

4 in 10

pull requests merged last month were opened by agents

227 of 527 in September · 1,069 of 3,311 since March

0

were merged by them

One repository: the main app monolith. Envato has 585 repositories; my domain owns 23 of them.

Speaker notes

Two numbers to start with, and a scope line so nobody over-reads them: these are from one repository, the main monolith behind the new app. Envato has 585 repositories and my domain owns 23. The pattern I'm describing runs across the domain, but the numbers on screen are the monolith's. Four in ten of the pull requests we merged last month were opened by autonomous agents. 227 out of 527 in September. Since March, when we switched them on, 1,069 agent pull requests have merged out of 3,311. It was 7% in March and 46% at the August peak. Not autocomplete. An agent picks up a ticket, works in its own environment, and opens the PR. And zero of them were merged by an agent. Every single one of those 1,069 was read, understood and merged by a person. Scope the zero honestly: it is zero since April. For five weeks in March and April our AI reviewer could approve, under a human's login. We took that away, not because it was wrong, but because we could not yet prove it was right. In a controlled pipeline those are the same thing. That second number is not a gap we are trying to close. It's the design. Hold on to it, because the whole talk is about how we keep it true while moving fast.

Agents write.
People own.

Speaker notes

That's the rule in four words. Agents produce plans, code, tests, PR descriptions, review findings. People own the intent, the judgement, the approval, and what happens in production. If you can't explain a change, you don't ship it. For those of you early in your career, this is the biggest shift: reading and reasoning about code becomes the core skill. Typing speed stops mattering.

PART 1

In my team's footsteps

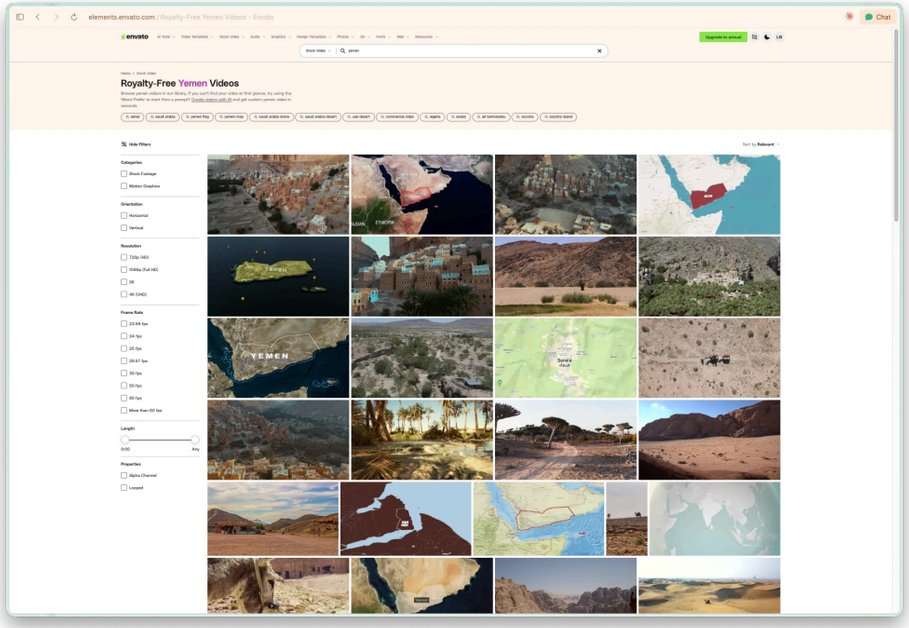
One real piece of work, six steps, start to finish

Speaker notes

Part one. I want you to walk in my team's footsteps through one real piece of work, from the moment a customer complained to the moment it was in production. Six steps: ideation, a spike, writing the tickets, building, review, and shipping. At each step I'll point at where AI sat and where a person decided. I chose this piece of work because it was one of our key search improvements for the quarter and because it runs end to end: a customer complaint, a model retrained on real queries, a new service in front of search, a production rollout, and a measured result. Two engineers built all of it in a few days. You can see every step in one sitting, which is rare for work of that size.

BEFORE

Search "Yemen". Get Romania.



Stock video · "yemen" · the results the customer saw

Speaker notes

The Envato App has millions of stock photos and videos, and people search for places a lot: travel content, news, backgrounds. Earlier this year a customer searched for a country and got footage from a different country. This is what they saw. Search for Yemen and the top rows are maps, country outlines, a flag, a 3D render of the border, and when you scroll, footage from somewhere else entirely. The real footage is there, but it is buried under things that merely contain the word. Keep this grid in your head. We come back to it at step six.

STEP 1 · IDEATION

Three conversations. One written decision.

Speaker notes

It started in a backlog review where the customer feedback came up. Then three short huddles over a week with two teammates. We're a remote, async-first company, so the writing is the work. Every meeting ends up as a note with decisions and actions. AI does the first pass from the transcript, and it pulls in context from older notes, tickets and Slack threads so I'm not starting cold. What I get from that isn't the summary. It's being able to ask "what did we decide about location detection in March" and get an answer with a source. The output of that week was one written state-of-play document for the team and product. AI drafted it. I rewrote every judgement call in it.

Precision over recall

A wrong "this is a place" costs more than a miss

Speaker notes

This was the decision that shaped everything after it, and it came out of a huddle, not a model. Our existing approach was an entity model that was very good at finding places and very bad at not finding them. It flagged "UTC" and "Piazza" as locations. If we route those queries to a different search path, we make search worse for everyone. So we agreed: we'd rather miss a location than wrongly call something a location. That's a product call. No agent made it.

STEP 2 · SPIKE

30% → 85%

precision, two days

Speaker notes

Then a spike. Two days to compare the existing model against a compact fine-tuned classifier that only answers yes or no: does this query contain a place? One of my teammates trained it. We tested on 20 thousand real queries, then 100 thousand. Precision went from around 30 percent to over 85, with similar latency. Where AI sat: it pair-wrote the notebook and the evaluation harness, and it wrote the queries that sampled real searches from our data warehouse. Hours instead of days. Where people sat: we picked the metric, and we set the time box. Two days to compare, two weeks to decide. Agents don't time-box themselves. I'll mention the model names since they're public: an entity model in the GLiNER family, versus a MiniLM binary classifier.

A model as the judge

Hundreds of queries, not twenty

Speaker notes

One more thing from the spike that changed how we work. We used to judge search quality by eyeballing grids of results. That caps you at maybe twenty queries before your attention goes. Instead we used a large language model as a relevance judge over a few hundred sampled queries. Two of us could evaluate a whole search profile in an afternoon. And the AI was useful as a sceptic. An earlier precision number for a bigger model had itself been judged by a model, and it asked the question we should have asked: is that number a red herring? It was worth checking.

STEP 3 · JIRA

A good card is the prompt

Context	why, what the spike taught us, links
Done when	outcomes you can observe
Resources	docs, dashboards, the eval set

Speaker notes

Then we turned it into work. Three cards: train and deploy the classifier, fix the worst country and city searches, and make the decision. Our stories follow a simple shape: context, done when, resources. Clear intent, not step-by-step instructions. We have a Jira skill for this. I give it a sentence and the spike notes, it reads the codebase, drafts the card with acceptance criteria and scope fences, and I edit it. Why this matters more than it used to: a cloud agent won't stop to ask a clarifying question. A thin ticket gives you a weak PR. The card serves the next engineer and the next agent equally. The bottleneck has moved from writing code to writing work.

STEP 4 · BUILD

Judgement stays local. Execution can travel.

LOCAL

routing logic
search profiles

CLOUD

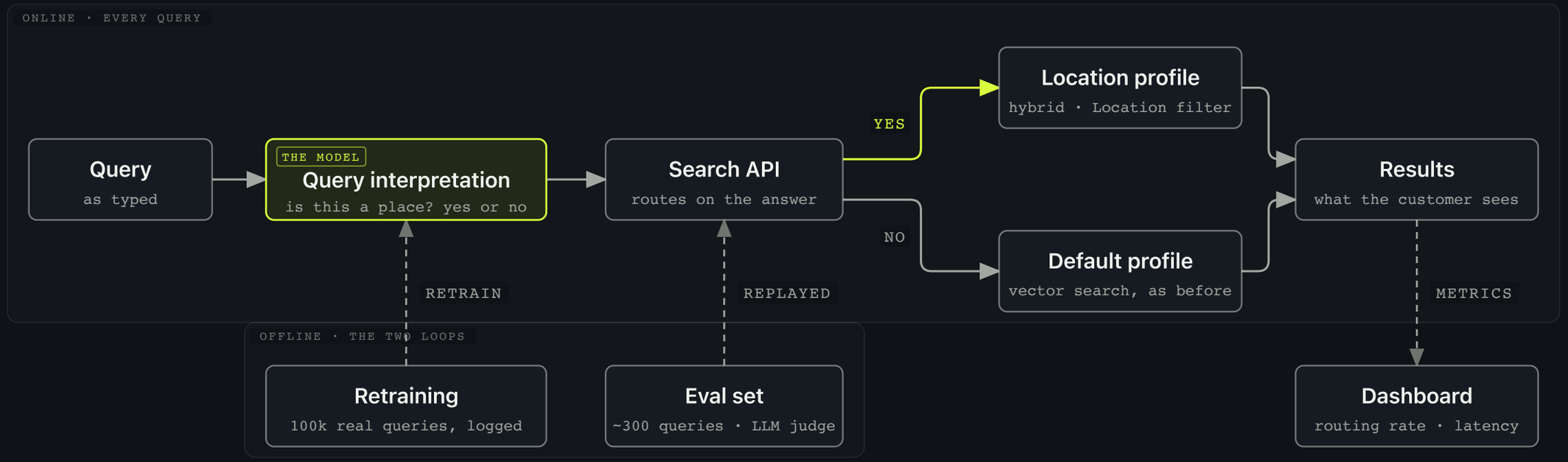
tests, wiring
config

Speaker notes

Building happened in two places. Locally, paired with an agent, I did the search routing logic and the search profile changes. Plan first, then code, then tests. Before I touched code, a second model reviewed the plan. In the cloud, I delegated the well-specified pieces straight from the cards: tests, wiring, a config change. A card gets a label, an agent boots an isolated environment with the full app running, does the work, and opens a PR with screenshots. It reads the same repo conventions my local session does. I reviewed those PRs like I'd review anyone's. The relevance judgement calls never left my machine.

What we built

Two engineers, a few days.
Precision 30% to 85%. No latency added.



Speaker notes

This is the shape of what the two of us built, so you can see it is not a small change. Top row, every query. The query goes through a new query interpretation service before it is searched. That service is the fine-tuned classifier from the spike, retrained on a hundred thousand real queries from our logs, and it answers one question: is this a place, yes or no. The search API routes on the answer. A yes goes to a new location profile, a hybrid of lexical and vector search with a Location filter. A no goes to the default vector profile, exactly as before. The classifier is in the BERT family and runs in milliseconds, and because it replaced an entity-extraction step and a language-detection step in the old pipeline, the end-to-end latency did not go up. Bottom row, the two loops. Retraining pulls real queries from the logs, which is how the model keeps up with what people search for. The evaluation set, a few hundred sampled queries judged by a language model, is replayed against the live API, so the same set that won the spike is the regression check in production. Routing rate and latency go to a dashboard, and I watched that during the rollout. Where AI sat: it pair-wrote the training notebook and the retraining pipeline, the service scaffolding, the routing code, the profile configuration, the evaluation harness and the dashboard queries. Where people sat: the decision to route on a yes-or-no answer rather than extracted entities, the precision-over-recall call, the profile design, and every review.

STEP 5 · REVIEW

Four reviewers.
One approver.

AI in CI · second model · walkthrough · teammate

Speaker notes

Review is where the zero comes from. Commits and PR descriptions come from skills, so they're consistent: problem, approach, evidence. For the big PR, a skill generated a one-page visual walkthrough so a reviewer can get the shape of the change in two minutes. Then the reviewers. An AI reviewer runs in CI within minutes and posts inline findings with a severity. A second model gives an independent opinion. And then a teammate reads it, reasons about it, and approves. The AI review is a required check, and the PR can't edit its own reviewer. I'll come back to that in part two. But the approval is human, every time.

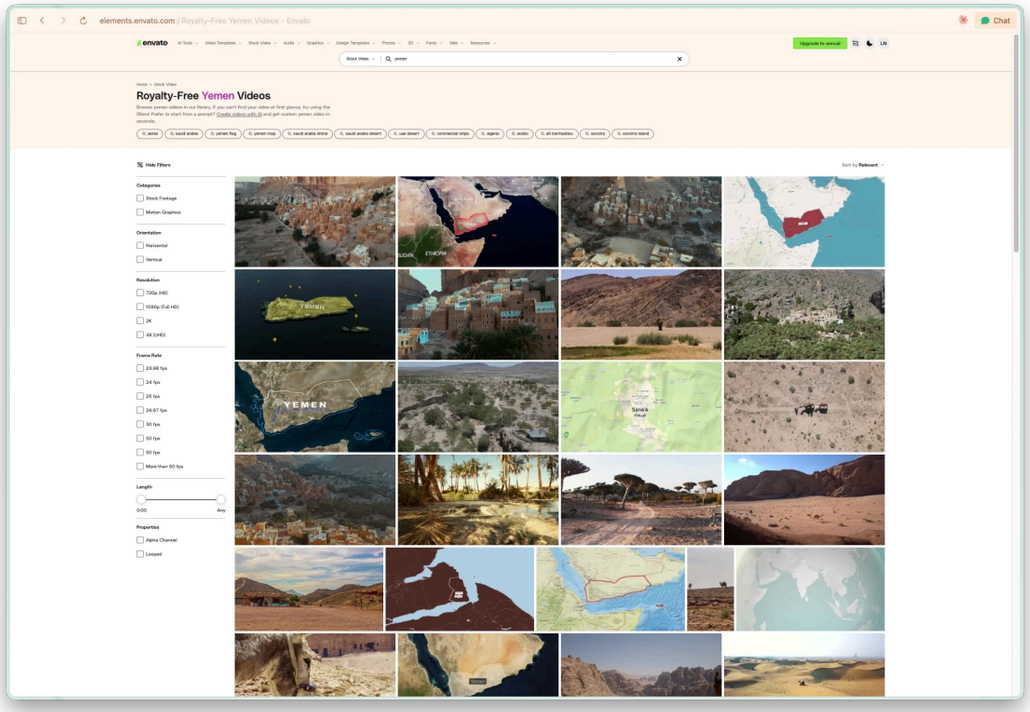
STEP 6 · SHIP

Ship. Watch. Search again.

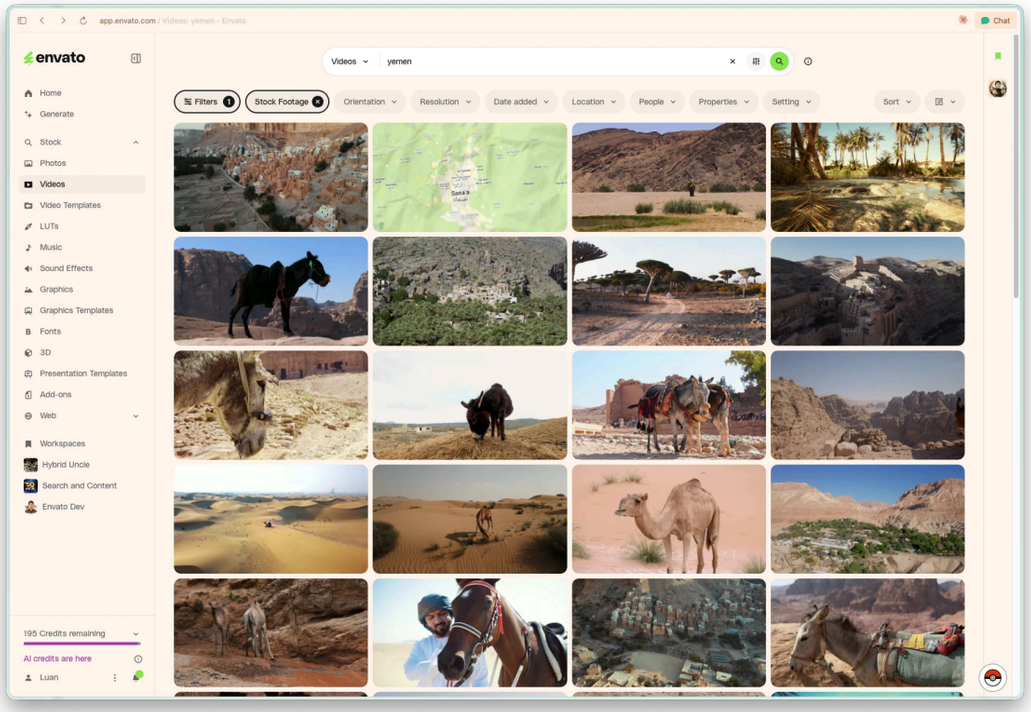
Speaker notes

Then we shipped. CI pipeline to staging, then production, behind a flag with a staged rollout. A person pressed the button. For monitoring, AI drafted the log queries and the dashboard: routing rate, latency percentiles, how many queries the classifier flagged. I decided whether the rollout continued. The best part was that the evaluation set from the spike became the regression check in production. Same few hundred queries, re-run against the live system. And then we searched for Yemen again.

AFTER



Before



After

Speaker notes

Same query, same catalogue. On the right the location profile is doing the work: towns, wadis, the dragon blood trees of Socotra, camels, people. The maps and flags are gone from the top rows, and there is a Location filter in the toolbar now. That is the whole piece of work, from a complaint to this grid. Everything in it had AI somewhere in the loop, and every decision in it had a person's name on it.

Two engineers.
A few days.

A retrained model, a new service in front of search, a production rollout. Precision 30% to 85%. No latency added.

Speaker notes

I want to sit on this for a moment, because it is the point of Part 1. Two engineers built everything you just saw, from scratch, in a few days: retrained the model on real queries, stood up the interpretation service, changed the routing, built the location profile, wrote the evaluation harness, shipped it behind a flag and watched it. Precision on the detection step went from around 30 percent to over 85, and the search did not get slower. A year ago that is a quarter of work for a small team, with the model retraining alone taking most of a sprint. The difference is not that the engineers got smarter. It is that AI was in every step, and the two people spent their time on the decisions instead of the typing. That is the honest version of the productivity claim. It is one project and one team, but it is a real one, with a before and after you can see.

Who decided

Stage	Person	AI
Ideation	direction	summaries, context
Spike	metric, time box	notebooks, data, judging
Jira	intent	drafting the card
Build	design, delegation	plans, code, tests
Review	approval	findings, walkthroughs
Ship	go / no-go	queries, dashboards

Speaker notes

That's part one in one table. Read the middle column top to bottom: direction, metric, intent, design, approval, go or no-go. The accountability never moved. The right column is wide. AI was in every step. But every row ends with a person. None of this works by accident, though. Part two is the setup underneath it.

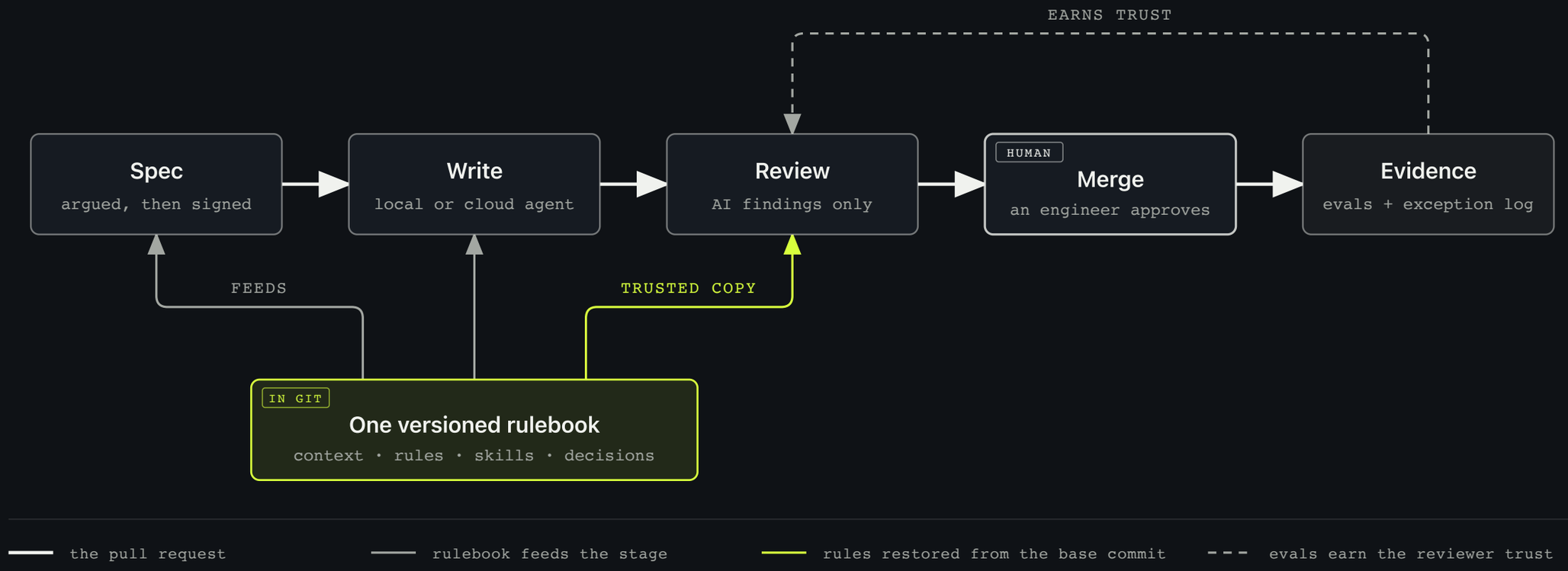
PART 2

What's underneath

Speaker notes

Everything you just saw depends on a few things being true in the repository and in the team. I'll go through them quickly.

One rulebook. Five stages.



Speaker notes

This is the shape of every piece of work, including the one you just saw. Spec, write, review, merge, evidence. One versioned rulebook feeds the spec, the writing, and a trusted copy goes to the reviewer. Evidence feeds trust back into review. Plant the second RSVP question here: AI doesn't merge. Every arrow into Merge has an engineer on it. Tech debt comes from decisions, not tools, and the decisions still belong to people. It comes back at the write, review and evidence stages.

RSVP Q1

Config is code

```
AGENTS.md           every assistant reads this
.agents/            the guidelines
.claude/skills/     the source of truth
.cursor/ .codex/    symlinks
```

Speaker notes

First: the AI configuration lives in the repo and goes through pull request review like any other code. One AGENTS.md at the root, which follows an open convention, so Claude Code, Cursor, Codex and Copilot all read it. The detailed guidelines live in a folder next to it. Skills follow the Agent Skills open standard, with one source folder and symlinks into each tool's directory. We use three tools on purpose. They have different strengths, and it keeps us from hard-wiring our workflow to one vendor. Press down for the full rulebook if anyone wants the layering in detail. This is the first half of the answer to RSVP question one, consistency between dev and production. We don't try to make everyone's model identical. We make the context identical and version it. Each config file carries a signature string and a script checks it, so "did the agent load the rules" is a test, not a hope. Behind the rules sit 208 architecture decision records the agents can cite, so they get a why, not just a what.

The rulebook, in full

Four layers. The agent loads the top two always, the rest on demand.

Layer	What	Loads
Always on	`AGENTS.md`, `CLAUDE.md`: identity, critical rules, pointers (~250 lines)	every session
Guidelines	17 canonical files: coding standards, architecture, testing, review checklist, CI, cloud, quality	by reference
Path rules	8 condensed rules, each pointing at its guideline	when a matching file is edited
Specialists	10 sub-agents with the full guideline embedded	by delegation

Path rule	Fires on
ruby-rails	`*.rb`, Gemfile, rake
rails-engines	`engines/**`, `use_cases/**`
rails-migrations	`db/migrate/**`
typescript-react	`*.ts`, `*.tsx`, tsconfig
testing	specs, tests, stories, cypress
ci-cd	pipelines, workflows, Dockerfiles
aws	`infrastructure/**`, CDK
quality	always (no path)

Plus three workflow rules for the QA kit, loaded only when its files are edited. Every file under CODEOWNERS; a script checks the signature strings.

Speaker notes

This is the full rulebook for anyone who wants to copy the shape. The point is not the content, it is the layering: the always-on part is small, and everything else arrives when the file you are editing says it should.

Load the minimum

Always on: identity and rules

By file path: domain rules

By task: skills

By delegation: specialists

Speaker notes

Second: context is finite and attention degrades, for models as well as people. The failure mode we avoid is a three-thousand-line instructions file the model half reads. Ours has four layers. Always-on is about 250 lines: who we are and the rules that never bend. Open a Ruby file and the Ruby rules load. Mention translations and the i18n skill loads. Need a deep dive on a Rails engine and a specialist takes it in a separate context and comes back with a result, not a transcript. Need it every session? Layer one. Tied to a file type? Layer two. Reference for a task? Layer three. Deep work with tools? Layer four.

A skill is a procedure.
A specialist is a colleague.

Speaker notes

Third, the two things you saw most in the story. A skill is a markdown file with a description of when it applies and a procedure. Our commit skill knows our seven rules for commit messages and the pre-commit checks. Our PR feedback skill fetches every review comment, classifies it and builds a task list. We have about 27 of them. A specialist is a sub-agent with the full guideline for its domain embedded, and its own tools. Rails engines, React routes, testing, CI pipelines. We also have a QA chain that drives a real browser: it writes a feature definition from the live app, designs tests, runs them, then goes exploring for bugs. The full list of skills and specialists is on the two slides below this one; they are in the PDF for anyone who wants to copy the shape.

Anatomy of a skill

```
---
name: commit
description: Creates git commits following the
  seven rules of great commit messages. Runs in
  an isolated context, so pass what changed and
  why. Example - /commit Added rate limiting
allowed-tools: Bash(git *), Read, Grep, Glob
context: fork
argument-hint: "<what changed and why>"
---
# Git commit skill

## Current repository state
- Status:         !`git status --short`
- Recent commits: !`git log --oneline -5`

## Procedure
1. Read the staged diff and the recent commits
2. Draft a subject line under 50 characters ...
3. Run the pre-commit checks; fix, do not skip
```

- This is our commit skill with the body trimmed. A skill is one Markdown file with a front matter block and a procedure.
- The description is the trigger. The agent reads it to decide whether the skill applies, so I write it as "when to use this", with an example call.
- The tools line is an allow-list. This skill can run git and read files. It cannot edit code, which is the point.
- `context: fork` runs it in its own context. It cannot see my conversation, so whoever calls it has to say what changed and why.
- Lines starting with `!` run a shell command as the skill loads, so the procedure begins with the real state of the repo rather than a guess.
- The body is steps and checks, written the way I would brief a new starter. Advice goes in the guidelines; a skill is what to do.

Speaker notes

I get asked "what does a skill actually look like" more than anything else, so here is one. The front matter does the routing, the body does the work. If you only remember one thing: the description is read by the model to decide whether to use the skill, so it has to say when the skill applies as well as what it does.

One folder, three tools

```
.claude/skills/commit/SKILL.md      source of truth
.cursor/skills/commit  ->  ../../.claude/skills/commit
.codex/skills/commit   ->  ../../.claude/skills/commit

.agents/ruby-rails-standards.md     the shared guideline
.claude/rules/ruby-rails.md         path rule: paths + source
.cursor/rules/ruby-rails.mdc        same rule, Cursor's format

script/validate-agent-config.sh     symlinks resolve,
                                    references exist,
                                    signatures present
```

- Skills follow the Agent Skills open standard, so one folder serves Claude Code, Cursor and Codex. The other two directories hold symlinks, so there is one copy to edit.
- Path rules are the exception. Each tool has its own format, so the rule is a thin file that names the paths and points at the shared guideline instead of repeating it.
- A script checks that every symlink resolves, every reference exists and every file still carries its signature string. I run it whenever I touch the rulebook; it has caught drift between the two editors' rule files more than once.
- Two kinds of skill live in that folder. Slash commands I call on purpose, like `/commit`. Reference skills the agent loads itself when the description matches what it is doing, like `translate`.

Speaker notes

We use three tools, so the layout matters more than it would for one. The rule I hold to is that anything that can be a symlink is a symlink, and anything that cannot points at a shared file rather than copying it. The validation script is what makes that a habit rather than a hope.

Skills · the daily habits

Slash commands and references, one source folder, symlinked into every tool. Names as they appear in the repo.

Skill	What it does
`commit`	commits in the team's shape; runs the pre-commit checks
`pr-description`	creates or updates a PR following the repo template
`pr-review`	review checklist plus the comment and approval workflow
`address-pr-feedback`	turns review comments into a task list, filters the noise
`pr-walkthrough-summary`	a designed one-page walkthrough of a large PR
`codex-plan-review`	a second model critiques the plan until it approves
`codex-code-review`	a second model reviews the branch, up to five rounds
`sdd-workflow`	spec to requirements, design, slices, tickets
`design-to-delivery`	splits an approved design into mergeable PRs

Skill	What it does
`jira`	read, create and update work items with templates
`dev-commands`	setup, services, database, common tasks
`agent-browser`	drive a real browser from the CLI
`translate`	i18n patterns and locale conventions
`failure-demand`	log a friction point, with time lost, to a shared sheet
`eval-ground-truth-review`	build the labelled dataset the AI reviewer is graded on
`create-specialist`	scaffold a new skill or sub-agent
`skill-creator`	write, improve and evaluate skills
`context-manager`	evolve the rulebook itself; which layer does a rule belong in

Plus 9 product-specific skills not listed here: generation model and eval setup, upload and download internals, local app browsing, QA kit sessions, internal publishing and marketing guides.

Speaker notes

Twenty-eight skills at the moment. These are the generic ones, the kind any team could copy. The nine I have left off are tied to our product internals, so they would not mean much outside the building. If you take one thing from this slide: half of these are about the pull request. Writing it, describing it, reviewing it, responding to review. That is where the habits were weakest and where codifying them paid most.

Anatomy of a specialist

```
---
name: ruby-rails-specialist
description: Reviews and implements Rails features:
  engine patterns, use case classes, migrations,
  API controllers, background jobs. Use proactively
  for complex Rails work across multiple engines.
tools: Read, Grep, Glob, Edit, Write, Bash,
  AskUserQuestion
model: inherit
---
# Ruby/Rails specialist

You are a Rails specialist with deep knowledge
of this codebase ...

<the full Rails guideline, embedded>
<the project overview, embedded>
```

- A specialist is a sub-agent. Same model, separate context window, a task in and a result out. My session never sees its transcript, only the answer.
- "Use proactively" in the description lets the main agent delegate without being told to. I use it for work that would otherwise drag a couple of thousand lines of guideline into my own session.
- The domain guideline is embedded rather than referenced. That is fine, because it costs the specialist's context, not mine.
- Each one gets its own tools. The QA chain's agents get browser tools because they test the running app. The Rails specialist does not need them.
- We scaffold these with a skill, so a new specialist takes minutes, comes out in the same shape as the others, and is registered in all three tools at once.

Speaker notes

The mental model that helped my team: a skill is something I do, a specialist is someone I ask. When the task is "go and understand this engine and come back with a plan", I want that reading to happen somewhere other than my context window.

Specialists · the colleagues

Sub-agents with a domain guideline embedded and their own tools. They run in a separate context and come back with a result.

Specialist	Domain
`ruby-rails-specialist`	engines, use cases, migrations, API controllers
`typescript-react-specialist`	routes, components, stories, i18n, design system
`testing-specialist`	RSpec, Vitest, Playwright, Storybook tests
`ci-pipeline-specialist`	build pipelines, caching, deployments
`file-upload-specialist`	the upload component and its patterns

QA chain	Step
`feature-definer`	explores the live page, writes the feature definition
`test-designer`	designs test cases from the definition
`test-execution`	runs the plan in a real browser, writes the report
`exploratory-tester`	goes off-script looking for bugs
`cypress-generator`	turns the plan into a runnable spec

Speaker notes

Ten specialists. Five are domain experts for our stack. Five form a QA chain that drives a real browser against the running app: define the feature, design the tests, run them, explore for what the plan missed, then generate a spec you can keep.

The QA chain, step by step

Five specialists, each handing a file to the next. All of them drive a real browser against the running app.

Step	Specialist	Reads	Writes
1	feature-definer	the live page, through browser tools	a feature definition: what the page does, with evidence
2	test-designer	the feature definition	a test plan with cases and expected results
3	test-execution	the test plan	a report, pass or fail per case, with screenshots
4	exploratory-tester	the definition and the report	a list of what the plan missed
5	cypress-generator	the test plan	a runnable spec we keep in the repo

A person reads the definition before step 2 and the report after step 3. The chain does not merge anything.

Speaker notes

This is the most "agentic" thing we run, and it is still a pipeline of files. Each specialist writes a document the next one reads, which means I can stop after any step, read what it produced, and correct it before the next one starts. The exploratory tester is the one that earns its keep: it is told to ignore the plan and go looking, and it finds the things nobody wrote a case for.

Rule, skill, specialist or hook?

I want to	I reach for	Because	Example
make sure the agent always knows something	an always-on rule	it is short and never bends	never commit secrets; use the use case pattern
apply knowledge only when a kind of file is touched	a path rule	it keeps the always-on part small	Rails rules load for `*.rb`
give the agent a procedure for a task	a skill	steps are easier to follow than advice	`/commit`, `/pr-description`
do deep work without flooding my session	a specialist	it runs in its own context and reports back	the Rails specialist, the QA chain
stop something whatever the model read	a hook	enforcement cannot depend on attention	pre-commit blocks ticket keys

The test I use: if it only matters when the model reads it, it is a rule or a skill. If it has to happen even when the model ignores everything, it is a hook.

Speaker notes

People ask which of these to build first. My answer is the bottom row. Start with the thing that must never happen, make it a hook, and then work upwards. Rules and skills make the agent better on a good day. Hooks are what you have on a bad one.

Your first skill

Start with the thing you have corrected the agent on twice

Write the description as "when to use this", with an example call

Write steps and what good looks like, not an essay

Allow-list the tools it needs and nothing more

Try it in a fresh session before you commit it

Keep it next to the code it describes

Speaker notes

This is the advice I give inside the team. The second correction is the signal: once I have told an agent the same thing twice, that is a skill waiting to be written. Fresh-session testing matters because the skill has to work without the conversation that made it obvious to me. And keeping it in the repo means it goes through review like code, and it rots in the same pull request as the code it describes, where someone will notice.

RSVP Q2

Hooks, not hope

format · lint · no ticket keys in comments · push guard · stop hook

Speaker notes

Fourth: a rule in a markdown file is a suggestion. A hook is enforcement. Pre-commit runs formatting and linting. It also fails the commit if a code comment or test name contains a ticket key, because a ticket key isn't a reason and the ticket will be closed by the time someone reads the code. Pre-push refuses to push to any remote outside the company, with no bypass flag. My favourite is the stop hook. When an agent finishes, a script checks whether the branch has meaningful changes and, if you've opted in, kicks off an independent review by a different model before you've even looked. This is the tech-debt half of RSVP question two. Debt comes from decisions and engineers still make them. What changed is volume: more plausible code per hour. So the guard rails went from advice to automation, and every one of these hooks traces to something an agent actually did. The ticket-key check exists because agents kept writing them. Nobody has ever praised a hook, so I lead with what they caught. The full hook list is below this slide. The other half is a friction log: a skill that writes what the AI cost us, and how long, to a shared sheet. That is how we know when AI is costing more than it gives.

Anatomy of a hook

```
// .claude/settings.json
"hooks": {
  "SessionStart": [{ "hooks": [{
    "type": "command",
    "command": "bash script/setup-worktree-env.sh",
    "timeout": 300 }] }],
  "Stop": [{ "hooks": [{
    "type": "command",
    "command": ".agents/hooks/code-review-stop.sh",
    "timeout": 30 }] }],
  "PostToolUse": [{
    "matcher": "Edit|Write|MultiEdit",
    "hooks": [{
      "type": "command",
      "command": ".agents/hooks/guidelines-drift.sh",
      "timeout": 10 }] }]
}
```

- A hook is an event, an optional matcher, and a shell command with a timeout. The editor runs it. The model is not consulted.
- The command gets JSON on stdin describing the event and answers with JSON on stdout: approve, block with a reason, or extra context to drop into the agent's transcript.
- Three events do most of our work. Session start bootstraps a fresh worktree with secrets and dependencies. Stop runs when the agent says it has finished. Post tool use fires after an edit, filtered here to the editing tools.
- The scripts live in one shared folder and are wired into the other editor's hook file too, under its own event names, so both tools get the same behaviour from a single script.

Speaker notes

The thing to notice is how little is here. Three events, three scripts, four timeouts. A hook is not a place for logic; it is a place to call a script that has logic, and that script can be tested on its own like any other shell script.

The stop hook, step by step

```
# .agents/hooks/code-review-stop.sh
[ "$ENABLE_AUTO_REVIEWS" = true ] || approve   # opt in
trap approve ERR                               # fail open
INPUT=$(cat)                                   # event JSON
grep -q '"stop_hook_active": *true' <<<"$INPUT" \
  && approve                                   # no loops

git diff --quiet origin/main...HEAD \
  && git diff --quiet && approve               # nothing new
reviewed_at_this_commit && approve             # already done

echo '{"decision": "block",
       "reason": "Changes on this branch have not
       had an independent review. Run
       /codex-code-review before finishing."}'
```

- Opt in by environment variable, because the review it triggers costs money and the first version shipped broken. Making it opt-in was the fix, and I would do that from the start next time.
- Fail open. Any error in the script approves the stop. A hook that can wedge the editor gets deleted within a week.
- Guard against loops. The editor tells the hook when it is already running because of a previous block.
- Only then look at the branch, and skip if the review already ran against this commit or in the last five minutes.
- Block with a reason. The agent reads that reason as its next instruction, so the sentence is written for the model, not for me.
- No network calls, under thirty seconds. The expensive part happens in the agent's next turn.

Speaker notes

This is the hook I like most and the one that embarrassed me. It shipped broken and nobody praised it when it worked. The order of the guards is the lesson: cheap exits first, the actual check last, and a reason the agent can act on.

The pre-commit, step by step

```
# .husky/pre-commit
export PATH="$HOME/.asdf/shims:$PATH"   # agents run
                                        # non-interactive
lint-staged                             # format, lint

./script/check-jira-jargon.sh           # ticket keys and
                                        # planning jargon in
                                        # comments and test
                                        # names, added lines


if staged apps/rails-api/; then
  bin/brakeman --quiet                  # security scan
  bundle exec packwerk check            # engine boundaries
fi

if staged '*.ts' '*.tsx'; then
  npm run typecheck                     # types
fi
```

- The jargon check is a grep over the staged diff rather than a linter plugin. One script covers Ruby, TypeScript, three AI tools and every human. Only added lines count, so old offenders never block a commit.
- It is anchored to our real ticket prefixes so SHA-256 and ISO-8601 pass. It skips a TODO that names its ticket. It has an escape hatch that prints a warning, so a bypass is visible.
- The heavy checks run only when matching files are staged. A front-end commit does not wait for a Rails security scan.
- The first line exists because agents and worktrees run hooks in shells where the version manager is missing. The check broke on a Mac, people bypassed it, and the fix was two lines of PATH.

Speaker notes

Why grep and not a proper linter rule? Because a custom RuboCop cop only covers Ruby, an ESLint rule only covers TypeScript, and neither catches a test description string the same way. An LLM judge is the wrong tool for a pattern-shaped mistake. The grep took an afternoon and has run on every commit since.

Hooks, in full

Three places a rule becomes enforcement: the agent session, the git client, and CI.

Agent session hook	What it does
`SessionStart`	bootstraps a fresh worktree: secrets, env, dependencies
`Stop`	session ends with unreviewed changes: nudge a second-model review (opt-in)
`PostToolUse`	edit a trusted rule file: reminder to update the reviewer's guidelines

Mirrored into the other editor's hook format, so both tools get the same behaviour.

Git and CI	What it does
`pre-commit`	format and lint staged files; block ticket keys and planning jargon in comments and test names; security scan; engine-boundary check; type check
`pre-push`	refuse any remote outside the company organisation, no bypass flag
`CI: AI reviewer`	required check on every PR; findings only, cites the rule it enforces
`CI: compliance tagger`	tags PRs that touch pricing, quantity or delivery code

Speaker notes

Each of these exists because of something an agent actually did. The ticket-key check exists because agents kept writing them into comments. The engine-boundary check got stricter because agents bypassed the module structure. The push guard exists because a private codebase and an eager agent is a bad combination. None of them has ever been praised in our Slack. All of them have caught something.

What the hooks taught me

Every hook traces to something an agent actually did

Fail open, finish in seconds, never call the network

Opt in when it spends money

One script, wired into every tool's hook file

Test on the laptop you ship from as well as in CI

Nobody praises a hook, so count what it caught

Speaker notes

Seven months of our team channel contain no praise for any hook and two complaints about ones that broke. That is the job. The ticket-key check exists because agents kept writing keys into comments. The engine boundary check got stricter because agents walked around the module structure. The push guard exists because a private codebase and an eager agent is a bad combination. I keep a tally of what each one has caught, and that tally is the argument for keeping them.

RSVP Q2

Spec first

product spec → requirements → design → stories → slices

A person approves each step

Speaker notes

Fifth: for anything bigger than a bug fix, we use spec-driven development. A product manager's Confluence page becomes a set of engineering files in the repo: requirements with testable acceptance criteria, a design with a decision log, stories, and tasks sliced so each PR is reviewable in one sitting. The format is compatible with other spec tools. Every phase has a human gate, and the approval is recorded with a name and a time. The agent records the approval. It can't invent one. When not to use it: small fixes, config changes, spikes. Vibe-code the spike. Use the spec to productionise it. Be straight about where it stands. Three epics have used it. The practitioner retro called reviewing the generated artefacts laborious, walls of text. Our engineering manager's verdict was that it isn't useful where there is ambiguity, discovery or unknowns, and we set no goals for it this quarter. It fits small, well-specified work. This is the second RSVP question, the one about losing the struggle of working out how. The thinking didn't disappear. It moved upstream. You argue with the plan before any code exists, a second model argues back, and your name goes on the approval.

Review the plan before the code

Speaker notes

Sixth, and this is the cheapest quality win we've found: the plan gets reviewed before any code exists. The agent checks its plan against how the codebase already does things. It runs its own review from six angles: security, performance, testing, operability, architecture, product. Then a different model reviews it, and they go back and forth until it approves. Different models have different blind spots. The bug in the plan costs almost nothing to fix. The same bug in production costs an incident. Our tooling lead's rule: never use the review model as the coding model. Keep the adversarial aspect.

RSVP Q1

Thin ticket, weak PR

What we learned running cloud agents

Speaker notes

Seventh: the autonomous cloud agents from the story. A Jira card or a Slack message triggers one. It boots an isolated machine with Postgres, Redis, the Rails app, Storybook and a headless browser. It reads the same configuration as a local session, does the work, and opens a PR with screenshots. The lesson after a few months is that input quality beats model capability. The agent won't pause to ask a question. Everything it needs has to be on the card. Honest rough edges: signed-in flows are still hard for it, and PR descriptions are weaker when the harness opens the PR rather than the agent. We talk about these openly. Its share of our merged pull requests went from 7% in March to 43% in September. Our review load did not go down. It went up, and it moved to the leads. We wrote that down too. The honest half of RSVP question one. In July we tried to move the agent to a repository-defined environment so it would be reproducible from the repo. The headless browser would not start in it, so the browser tests broke the same evening. We reverted to the managed snapshot that night and amended the decision record: reproducibility traded for working tests, one evening. The bigger operational cost has been the vendor's metering: four spend-cap outages and one API break in seven months. The agent runs on a third party's meter.

One cloud agent run, start to finish

Stage	What happens	Who
`Trigger`	a label on a Jira card, or a chat message naming the agent	a person
`Boot`	a machine from a snapshot: Ruby, Node, Postgres, Redis, a headless browser. A start script brings the services up and catches the database up on any new migrations	the harness
`Read`	the same AGENTS.md as a laptop session, plus one guideline about this environment and its quirks	the agent
`Work`	a branch, commits through the commit skill, the same hooks on every commit	the agent
`Open`	a pull request from the repo template, with screenshots from the browser	the agent
`Review`	the AI reviewer in CI, then an engineer	both
`Merge`	an engineer	a person

The environment guideline exists because the agent cannot ask. Everything a new starter would ask on day one is written down for it.

Speaker notes

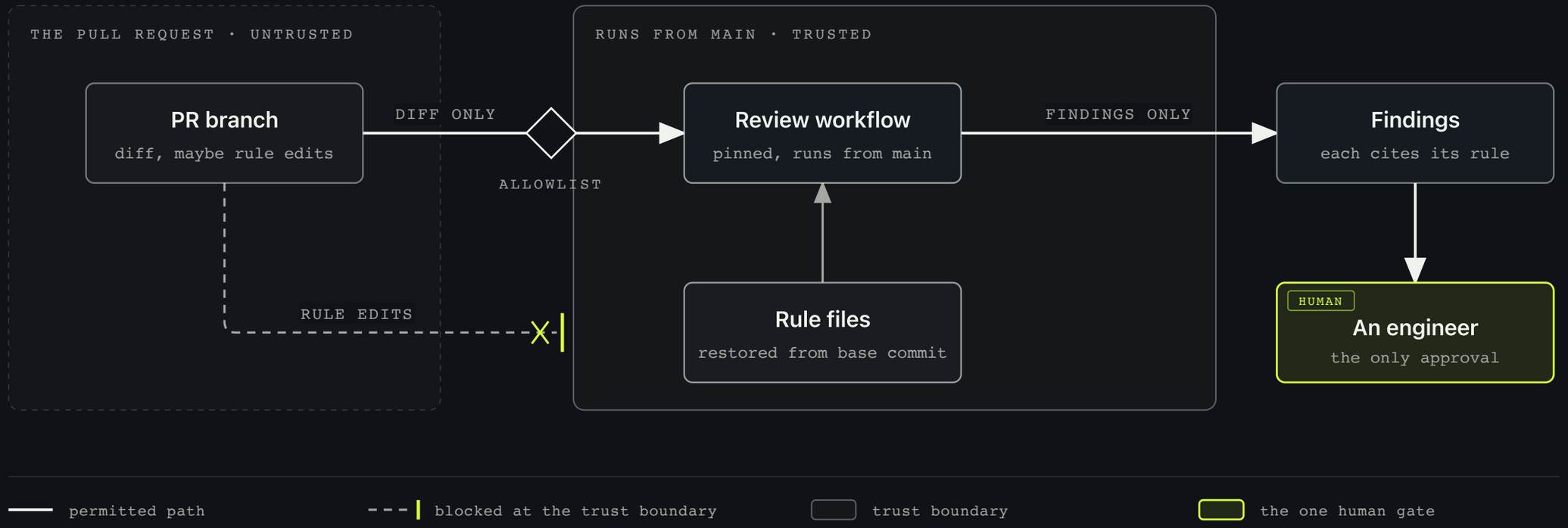
The part people underestimate is the boot. A cloud agent is only as good as the machine it wakes up on, and that machine has to be ready without anyone watching. We learned to split "install once" from "start every time", to fingerprint the migrations so a stale snapshot catches itself up, and to write the environment's quirks into a guideline the agent loads only when it is running in the cloud. None of that is AI work. It is the plumbing that made the agent usable.

The PR can't edit its own reviewer

Speaker notes

Eighth: the AI reviewer in CI, and why we trust it. An AI reviewer is only trustworthy if the author of a PR can't influence the review of their own diff. So the prompt, the schema, the scripts and the shared guidelines live in a separate repository that PR authors can't write to, pinned to a tagged release. The consumer repo's guidelines are restored from the base commit before the model runs, so a PR can't change them either. A PR that tries to weaken a shared rule is itself a top-severity finding. It took three decisions over four months. March: a reviewer action inside the repo with structured output, because the vendor's connector could only react with emoji. May: extracted to a shared workflow. Our wrapper is 52 lines of configuration, down from roughly 600 lines of review machinery, with a bot identity so an approval can never be mistaken for a human's. June: on every PR. It posts inline findings with three severities and a structured verdict, and every finding cites the rule behind it. On code paths with financial impact, uncovered changes are automatically blocking, which is what makes it acceptable to our auditors. It does not approve. Findings only. It did approve for five weeks in March and April, under a human's login, and we took that away until evals could earn it back. Both RSVP questions land here. Consistency: the reviewer is pinned, and it reads the same rule files the coding agents read. Debt: a shortcut that bends a rule gets caught at review with the why attached.

Where the trust boundary sits



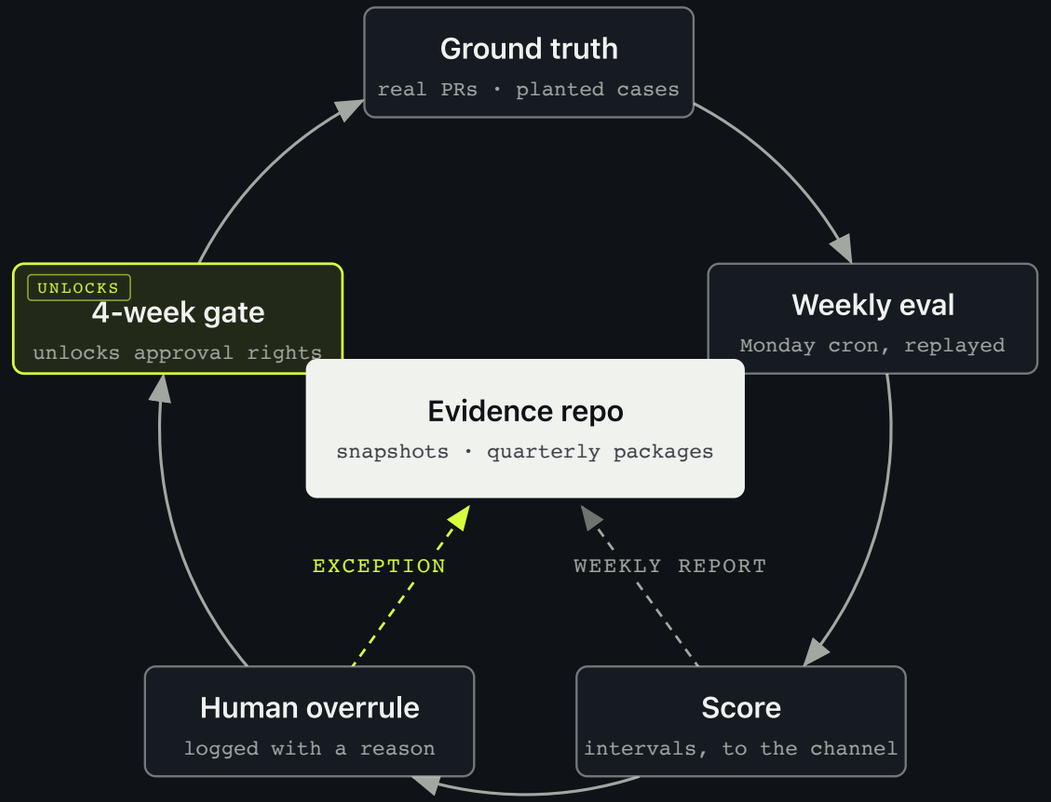
Speaker notes

The same thing as a picture. The pull request is on the untrusted side. Only its diff crosses. The workflow runs from main, reads rules restored from the base commit, and anything the PR did to its own rule files is stopped at the boundary. Findings go back. The only arrow into approve is the engineer.

RSVP Q2

Who reviews the reviewer?

30 September 2026:
no eval week has passed yet.
The four-week clock has not started.



the weekly loop · written back to the evidence repo · a human overrule, logged · the gate to approval rights

Speaker notes

If a bot is going to gate merges, something has to grade the bot. A ground-truth dataset: real PRs that were later reverted versus clean merges, plus planted scenarios like a pricing bug, a missing authorisation check and a prompt-tamper attempt. A cron job every Monday replays it through the reviewer and posts the score with confidence intervals. When a human overrules the bot, an issue form records the direction and the reason. Monthly snapshots, because issues are mutable, and a quarterly evidence package into a tamper-evident repository, where missing weeks are marked missing rather than hidden. This closes RSVP question two: that exception log is a tech debt register by another name. Decisions made visible. Now the honest state, and I say this before anyone asks. The ground-truth dataset landed mid-September. No eval week has passed its thresholds yet, so the four-week clock to approval rights has not started. Two exceptions have ever been logged, both in the first fortnight. The mechanism exists. The habit does not.

Local and cloud, same brain

Speaker notes

Ninth, how it runs day to day. Locally it's Claude Code, Cursor or the Codex CLI on your machine, with permission prompts, and connectors into Jira, Confluence, GitHub, Figma and our logs. That's where judgement happens: pairing, planning, reviewing. In the cloud it's an isolated machine per task, built from a snapshot, triggered by Jira or chat. That's for execution-heavy, well-specified work. Both read the same configuration. We even keep a guideline file that tells the cloud agent about its own environment's quirks.

Tokens are cheap.
Attention isn't.

Speaker notes

On cost. We're on enterprise plans for Claude, Cursor and Codex, per seat. The usage limits are generous enough that nobody rations tokens for daily work. The visible cost is cloud agent runs: isolated machines plus frontier models. The way I'd frame it for you: a wasted agent run costs a few dollars. A badly reviewed PR costs an incident. So we spend on slicing, walkthroughs and review tooling, not on cheaper models. And the honest cost is attention, not tokens. Our engineering manager posted last month about engineers having to manage agents without being taught how, and told people it is fine to let agents idle and cap how many streams they run.

Prompts rot like docs

Speaker notes

Tenth: keeping all this alive. Skills and prompts rot the same way documentation does. Two things keep ours fresh. They live next to the code they describe, so drift shows up in the same PR. And people use them every day, so a broken skill gets noticed within hours. We have a skill for writing skills, and one for scaffolding specialists. Shared skills ship through an internal plugin marketplace, one plugin per domain, maintained by a community of practice. When something's annoying, we log it, and that log drives what we fix next. For the review prompt that gates merges, every material change has to pass a batch evaluation first. We treat it with the rigour of authentication code.

A software factory

card → scoped → planned → built → reviewed → merged

Speaker notes

Where this is going. Another team at Envato is building what they call a software factory, and we're early consumers of it. It generalises the cloud agent pattern. You label a card. A scoper works out what's missing and proposes a plan. A person approves the plan on a dashboard. Workers build it. If a step blocks, a replanner adjusts rather than stopping. In a few allow-listed repositories, if every check is green and no human has commented, the factory can merge its own PR. Our app isn't one of those yet, by choice. So the zero will move for low-risk repos. The principle won't: someone can explain every change.

What the team said

> "Reviews are very much still required, perhaps even slightly more so."
>
> lead engineer · month one

> "How does it notice the rule when reviewing but not when generating?"
>
> principal engineer · month four

> "I only use it when the requirement is super clear."
>
> senior engineer · month five

> "It's not useful if there's ambiguity, discovery or unknowns."
>
> engineering manager, on spec-driven development · month five

> "Engineers now have to manage agents, but haven't been taught the coping mechanisms."
>
> engineering manager · month seven

Speaker notes

Five quotes from seven months of our team channel, by role and month of the rollout. I read them rather than paraphrase them. Let the last one sit for a beat. Engineers now have to manage agents, and nobody has taught them how. That is the gap I think about most.

What is not done

- × The evals are not passing yet
- × The overrule habit has not formed
- × Review load went up, and moved to the leads
- × One engineer owns the plumbing
- × The agent runs on a third party's meter
- × One production incident in seven months

Speaker notes

The honest gaps. The evals have not passed a week, so the reviewer still cannot approve. The overrule log exists and almost nobody uses it. Review load is up and it sits on the leads. One engineer owns the plumbing, which is a bus-factor problem I have not solved. The agent runs on a vendor's metering, which is where most of our outages came from. And one production incident attributed to an agent PR in seven months: an admin lockout, reverted the same day, and it exposed a latent auth gap we did not know we had. The failure I usually admit to, the July environment, lasted one evening. The one I should admit to is that the eval meant to earn the reviewer's trust has not produced a passing week yet.

Start Monday

Write the spec before the prompt

Read every line you ship

Put your conventions in the repo

Slice for a one-sitting review

Hooks, not hope

Get a second opinion

Speaker notes

None of this needs an enterprise licence. Six things you can start on Monday. Write the spec before the prompt: context, done when, resources. Read every line you ship, and don't approve what you can't explain. Put your conventions in a file in the repo; an AGENTS.md is an afternoon's work. Slice work so a person can review it in one sitting. Automate the guardrails. And get a second opinion, from another model and then from a person. If I had to pick one, it's the first. Writing clear work is now the highest-leverage skill in engineering, and it's the one we teach least. And one more for the people who asked the RSVP questions: version your rules, and feed the same file to your reviewer that you feed to your coding agent. Engineers still make the decisions. We just made the decisions visible.

4 in 10 written by agents. 10 in 10 understood by people.

Thank you · Luan Nguyen · Envato

Speaker notes

That's the talk. Agents write. People own. Happy to take questions on the trust boundary, the cloud agents, or how to start with a small team.