The DevOps Survival Guide



Practical DevOps in the Age of AI Agents

DEVOPS DAILY

The DevOps Survival Guide

Practical DevOps in the Age of AI Agents

How to stay sane, ship fast, and sleep again

DevOps Daily

The DevOps Survival Guide: Practical DevOps in the Age of AI Agents
First edition, 2026. Technical facts checked in October 2026.

Copyright © 2026 DevOps Daily.

The text and illustrations are licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0), creativecommons.org/licenses/by-nc-nd/4.0. The code examples are licensed under the MIT License.

The people and companies in the scenarios are fictional. Product names such as Kubernetes, Terraform and GitHub are trademarks of their owners; their use here does not imply endorsement.

This book is provided as is, without warranty of any kind. Test every procedure in an environment where a failure is safe before you use it in production.

Errata and the current edition: devops-daily.com/books/devops-survival-guide. Questions and corrections: hello@devops-daily.com.

Introduction

It's your first morning. Maya from development sends you the deployment README. It says "ask Alex." Alex from operations is in a meeting, and you're still figuring out how to log into the monitoring system.

Meanwhile, the coding agent has opened three pull requests since breakfast. One of them changes the deploy script nobody understands, and it's waiting for a reviewer. Your name is on the list.

Congratulations. You just got a DevOps job, and you're not the newest contributor on the team.

Maybe you're coming from operations, or you're a developer who got tired of wondering why code explodes in production. Maybe you're new to tech entirely and DevOps seemed like the right way in. Or maybe you've done this work for years, and the job changed shape around you: more changes to review, more access to grant, and a teammate that works at machine speed and rarely says "I'm not sure."

Whatever brought you here, you're probably feeling a mix of excitement and low-key panic. There are so many tools and acronyms, a new AI product every week, and everyone on the internet seems to know exactly what they're doing.

Take a breath. I wrote this book to help you find a manageable place to start.

What Changed, and What Didn't

If you're starting a DevOps job now, or your current one just changed, you're working next to agents. They write code, open pull requests, run

in CI, call tools and investigate incidents. DORA's 2025 research found that about 90% of the technology professionals it surveyed use AI at work. Its main finding was that AI amplifies what a team already is: good practices get stronger, and weak ones fail faster.

Treat any single number about AI and productivity with care. In a 2025 study by METR, sixteen experienced open-source developers, working in large repositories they knew well, took 19% longer with early-2025 AI tools, while believing afterwards that the tools had made them about 20% faster. That says nothing direct about you, today, with today's tools. It does say that feeling faster and being faster are different measurements.

Here's what I think changed. Producing code got cheap. Reviewing it, running it, recovering it and explaining it did not. A person who can read a diff, a plan and a trace, and who knows when to stop and ask for help, is exactly who a team needs next to an agent. Senior engineers now spend more of their time framing work, reviewing it, and writing the context files that agents read. I can't promise what any of this means for your career. I can tell you which skills this book practices: the ones that matter more when code is cheap.

And here's what didn't change:

- Small batches are easier to review and to recover.
- A rollback doesn't undo data.
- Users judge the outcome, not your dashboard.
- Someone owns each decision.
- People need sleep.
- You don't run what you can't explain.

Every AI section in this book applies that list to a new kind of contributor. Chapter 13 brings it together into an operating model for agents.

The tools change fast. Some product names and settings in this book will date quickly, and your team may change assistants more than once while you work there. That's normal, and it isn't the part to worry about. The skills under the tools change much more slowly: finding a fault from the evidence, reading a system end to end, owning an incident until it's closed, knowing what a change costs, and checking work you didn't write, whoever or whatever wrote it. Most of this book is about that slower part. The tools in it are examples; the habits are the point. If you want a plan for building those habits on purpose, "Work That Outlasts the Hype," the short section after chapter 14, gives you one for your next 90 days.

What You're Walking Into

Your new job description probably mentioned "automation," "cloud infrastructure," and "CI/CD pipelines": the workflows that check code changes and prepare or deploy releases. It probably talked about improving deployment processes and helping development and operations work together. These days it may also mention "AI-assisted workflows."

What it probably didn't mention is that you'll spend your first month figuring out which of the two hundred alerts in your monitoring system matter. Or that the deployment documentation leads you to someone who isn't available. Or that the "automated" infrastructure includes a bash script from before anyone on the current team joined, which nobody understands but everyone is afraid to touch.

It also didn't mention the newer problems. An agent can write a pull request in minutes, so the review queue grows by the hour. A tool server runs with a token nobody remembers scoping. The agent instruction file still names a build command that was removed months ago. None of these are exotic. They're the old problems of ownership,

access, review and documentation, arriving faster.

This book starts with those everyday problems and helps you decide what to tackle first.

What This Book Is (And Isn't)

I've written a practical survival guide for the problems you're facing right now. How do you improve a deployment process that everyone's afraid to change? How much may an agent do before a person approves? Who reviews the code that nobody typed? How do you drive change without a fancy title?

I explain ideas as they come up in practical problems. This book won't teach you every DevOps tool or replace the documentation for your stack. It will help you ask better questions about the systems you've inherited.

The human side of the work matters throughout. Technical problems involve trade-offs, and sometimes the right move is to change a requirement or raise a constraint with someone who can decide. Building trust helps you have those conversations. Knowing your limits helps you keep doing the work without burning out.

Who This Book Is For

This book is for engineers doing DevOps work now that agents are part of the team. Its center is the first few years, roughly 0-3 years of experience: you might be brand new, or you might have been at it for a while but still feel like you're making it up as you go. The scenarios follow a new hire, because a new hire sees what everyone else has stopped noticing.

If you've done this work for longer, the agent material is written for you too. Experienced engineers are usually the ones who review agent changes, decide what an agent may touch and own what it ships. Chapters 3, 7 and 13 are a good place to start.

You don't need years of experience with Kubernetes or Terraform. You do need to be willing to learn and improve how you and your team work.

If you're a manager or team lead, this book can help you set the rules your team and its agents work under, and show you what a newer colleague is dealing with and how to support them. Developers and support engineers can use it to understand the operational problems behind the requests coming their way.

How This Book Works

Chapters 1 through 14 each focus on a challenge you're likely to face. Each opens with a short scenario, followed by practical advice, and ends with "Your Next Steps." Those actions fit within an ordinary paid week. The groups "Do this yourself," "Propose this," and "This needs an owner" distinguish what you can do from what needs agreement or an organizational decision. Use the fallback when you lack access, budget or authority.

You can read front to back, starting with your first week and working through deployments and day-to-day operations before reaching organizational change, agents and sustainable work. Or jump to the chapter for the problem you have today: chapter 3 for deployments, for example, or chapter 6 for on-call. Follow cross-references when you need more context, then pick one action to try.

A few rules come up in many chapters. I state each one fully once and point back to it after that: the escalation rule lives in chapter 6, and the

rules for agents live in this introduction and chapter 13.

I assume you can find your way around a Linux shell, use Git to inspect and record changes, and navigate one cloud console. You don't need expertise in a particular provider. If one of those foundations is new to you, keep its introductory documentation nearby and practice in a learning environment you are allowed to use.

The map groups the chapters by topic, with arrows tracing the front-to-back reading order.


Survival
You Just Got Here
1
2
DevOps Is Not a Job Title
The Deployment Mess
3
Operations
4
Infrastructure That Doesn't Hate You
Monitoring vs. Guessing
5
6
On-Call Without Trauma
Security Is Your Job Too
7
8
No One Reads Your Docs
Strategy
Platform Thinking
9
10
Tech Debt, But Make It Boring
Measuring What Actually Matters
11
12
Driving Change Without a Fancy Title
Agents and People
The Agent Has Production Access
13
14
DevOps That Doesn't Burn You Out

The People You'll Meet

The scenarios follow one small company and the people you'll work with there. Alex is the senior operations engineer: calm in a crisis, keeper of a backup ritual you'll hear about in chapter 1. Maya is a developer who automates anything she has to do twice. Sam is a developer who ships fast and has opinions about it. Nora is the database engineer who remembers the last rewrite. Chris is the engineering manager, balancing delivery and reliability, preferably before the end of the day. Eli follows the cloud bill until it has an owner. And then there's the coding agent: fast, eager, confident, and sometimes wrong.

You're the new hire.



A Note About Tools

I've kept this book tool-agnostic as much as possible. The principles

apply across different deployment tools and hosting environments. When I mention a specific tool, it's an example you can compare with your own setup. Your tools will evolve, and your team may change stacks. Learning how to check a proposed change and understand its consequences will remain useful.

How to Use AI While You Read This

I use the same workflow throughout this book. First, I check what I may share with an assistant: public concepts freely, company code and configuration only where policy allows, never credentials, and customer data only through tools approved for it. The exercises in this book use synthetic or redacted material. I treat every answer as a hypothesis and check it against documentation, a test, or a person before acting. I don't merge or run anything I can't explain, which in practice means knowing its effects, the permissions it uses, and the way back.

When an agent can act, I sort what it does into three kinds. Investigation, which is read-only, it may do on its own. Bounded, reversible, tested actions that the service owner approved in advance, it may also take on its own, under a rate limit, with every action logged and a stop switch that works without the agent. Everything else waits for a person's approval. Chapter 13 explains where those lines go and how they move.

For example, ask an approved assistant to explain a deployment configuration you are permitted to share. It might claim that every push deploys to production while overlooking a branch condition. Check that claim against the configuration and the tool's documentation; ask a teammate about anything still unclear. Don't turn an unverified explanation into operating instructions.

One habit pays off in every chapter: when something breaks in a safe

environment, diagnose it yourself first, then compare your answer with the agent's. You learn the system and the agent's blind spots at the same time.

Check which tools and uses your organization permits. You may be allowed to work with synthetic examples even when company material is off limits. If policy forbids AI use, skip the marked AI sections. The principles and core exercises do not depend on AI.

What You'll Get Out of This

I want you to leave with a way to choose your next useful improvement and discuss it with the people affected. You don't have to fix everything at once. A small change you can verify is a good start.

The scenarios draw on situations I've encountered or heard about from other engineers, with names and details changed. Use them to recognize a problem, then check how your own system differs.

At the back, the Glossary gives short working definitions. Further Reading points you toward sources for deeper study and lists the sources for the dated facts. About This Book has edition details and tells you how to report an error or send me a question.

Your First Three Moves

1. Write down one problem you're facing today. If you haven't started the job yet, choose one from the opening scenario.
2. Find the chapter that addresses it, or start with chapter 1 if you're still getting your bearings.
3. Choose one "Do this yourself" action from that chapter for next week. If access is missing, use its fallback and record what you need to ask for.

The Agent Has Production Access

Maya opens a pull request just before lunch. Nine hundred lines changed across Terraform, Kubernetes manifests and the deployment pipeline.

"That was quick," you say. She only picked up the ticket this morning.

"I gave it to the coding agent," she says. "The tests pass."

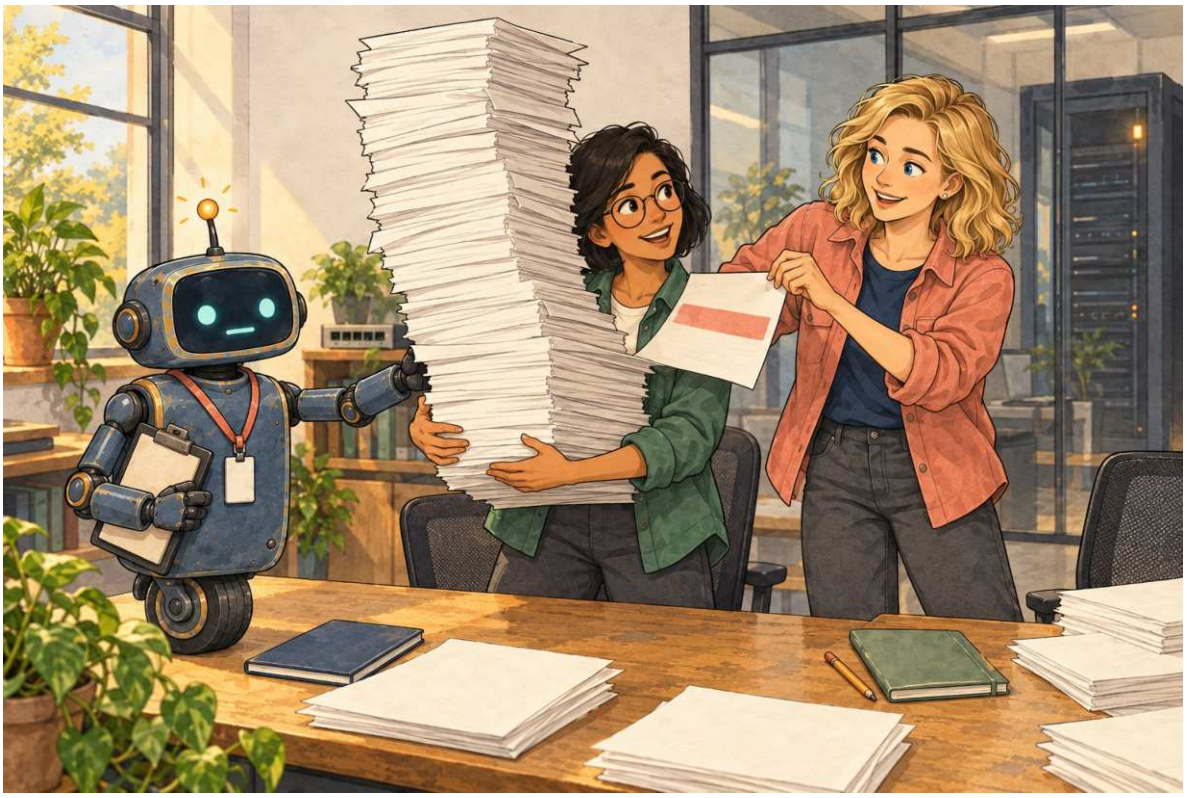
The description is polished. The new design has autoscaling, a queue, retries, dashboards, and three paragraphs about fault tolerance. The plan output has one line that deserves more attention than the other nine hundred: the production database will be destroyed.

You ask Maya whether that's expected.

She reads the line again. "No. Definitely not."

The agent renamed the database resource "for consistency" and assumed nothing important lived at the old address. The plan destroys the database at the old address and creates a new, empty one at the new address. The pipeline is green because syntax isn't intent.

Nobody did anything reckless. Maya used an approved tool in the normal way, and the agent produced a plausible change. The failure was in the operating model around it: what the agent was asked to do, what it could touch, and what had to happen before its work reached production. That operating model is what this chapter is about.



The Job Changed Shape

DevOps engineers have always translated intent into running systems. A team needs a service, and you turn that need into infrastructure, delivery, observability, access and recovery. Agents shorten the translation step. They can draft all five before your coffee cools.

That speed is useful, and it moves the bottleneck. When a hundred lines of configuration took half a day, the author absorbed context while writing: they looked up the provider behavior, noticed the resources next door, and made small decisions along the way. When an agent writes the same hundred lines in a minute, none of that understanding comes with them. The reviewer gets more code with less shared reasoning behind it.

So the work that remains is the work around the code. I'd name four skills: framing the outcome and its constraints, supplying the right

context, verifying the behavior, and owning the result after the agent stops. None of them is a prompt trick. A clever prompt can't rescue a vague requirement, a missing test environment, or a team that merges changes nobody can explain.

A Working Loop for Agent Changes

I treat an agent's change like one from a fast contributor I haven't worked with yet. I give it useful work, limit what it can touch, review what it produces, and never confuse confidence with evidence. The loop has six steps.

Frame. Start with the outcome and the boundaries. "Add autoscaling" is weak framing. "Keep checkout p95 latency under 500 milliseconds during a traffic spike of three times normal. Scale between three and twelve replicas. Don't change the database, the public API or the network. Alert when the maximum replica count holds for ten minutes" is a job the agent can do and you can check. Say what it must not touch and how you'll know it's finished.

Constrain. Give the agent the least access the task needs. Read-only repository access is enough for analysis, and a branch is enough for a change. Production credentials are almost never needed to draft a pull request. Separate capabilities that one tool might bundle: reading logs is different from restarting a service, and opening a pull request is different from merging it. A Terraform plan is different from an apply, but a plan still runs code: providers execute, and an `external` data source usually runs its program during the plan, with the plan's environment. Plan an agent's change in an isolated job with read-only credentials and limited network access.

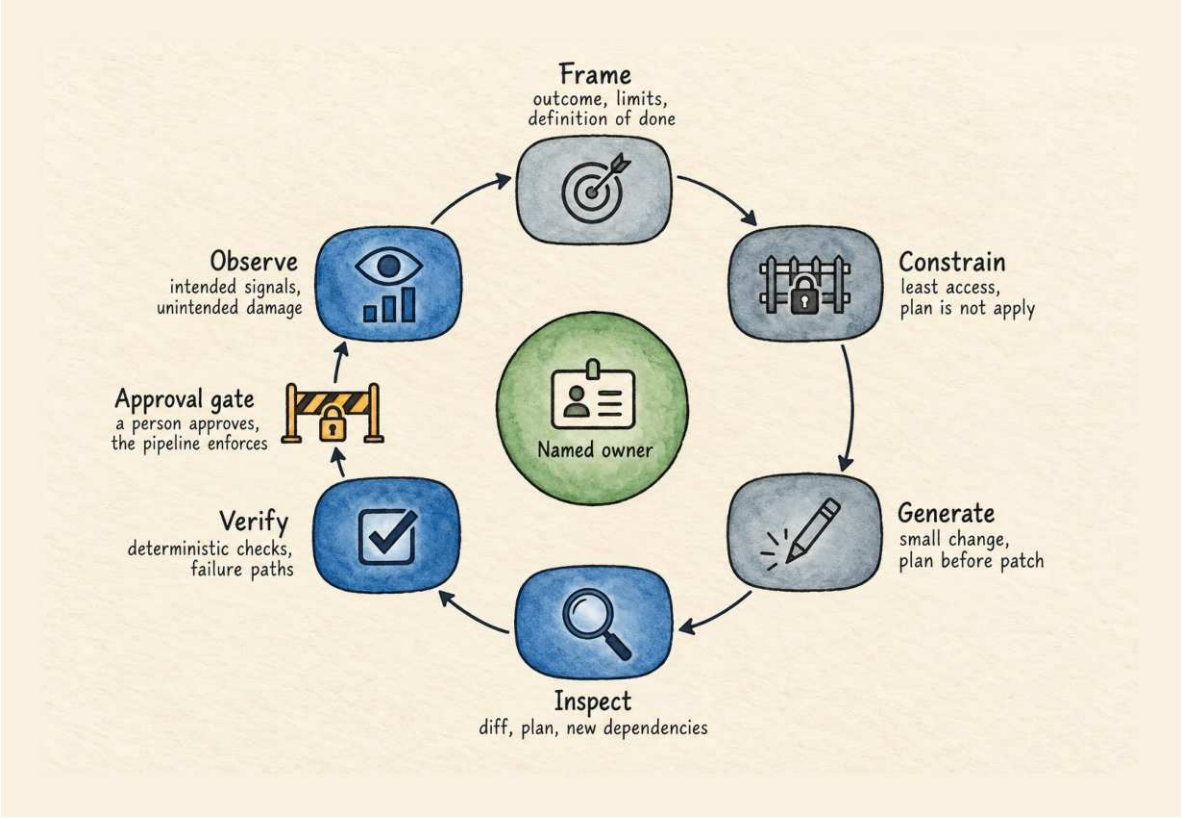
Generate. For work with a large blast radius, ask for a plan before a patch; the plan shows its assumptions while they're cheap to correct.

Keep the change small. An agent is perfectly capable of combining a dependency upgrade, a module refactor, a naming cleanup and a feature into one elegant disaster. Batch size still matters when the batch took seconds to produce.

Inspect. Read the diff, the generated files, the lockfile changes and the plan. Look for deleted controls, widened permissions, changed defaults, new dependencies and resources marked for replacement. Ask the agent to explain its choices, but don't treat the explanation as proof; a model can write a tidy rationale for a mistake. Evidence comes from the repository, the tests, the plan, the documentation and the running system.

Verify. Use deterministic checks wherever you can: compilers, linters, tests, policy engines, scanners, plans, schema validation and dry runs. They are less charming than an AI review and much better at saying the same thing twice. Test the failure paths too. What happens when the dependency times out? Can the migration go back? Is the write idempotent? Does the service still start when the model provider is unavailable?

Observe. After release, watch the signals tied to the outcome you asked for, and the ones that show damage you didn't ask for: error rate, saturation, support tickets. If your team labels agent-assisted changes, use the label to answer one question, not to blame the tool: are these changes improving throughput without raising rework and instability?



A person who understands the change approves it, and the pipeline enforces that rule rather than trusting everyone to remember it. Chapter 3 shows how: required reviews, code owners for the risky paths, and workflows that don't run on an agent's pull request until someone has looked.

The Plausible Wrong Answer

Back to Maya's pull request. She asks the agent to explain the destroy line. It answers: "The plan only renames the database resource. Terraform updates it in place, so no data is lost."

That's plausible, and it's wrong. Read the plan, not the explanation. It shows the database at the old address being destroyed and a new one being created at the new address. Nothing is updated in place. Renaming a resource changes its address. Without a `moved` block, Terraform sees one resource that disappeared and a new one to create,

as chapter 4 explains.

Maya's first fix is a `moved` block from the old address to the new one, followed by a fresh plan that shows the database moving to its new address with nothing destroyed. Her better fix is to ask whether the rename was needed at all. It wasn't, so she drops it, and the pull request shrinks by a few hundred lines. The team also adds a policy check that fails any plan that replaces or deletes a database without a named approval. That check will still be there when the agent's next explanation sounds just as sure of itself.

Context Is Part of the System

An agent works from the context it can see. That makes repositories, runbooks, decision records, service catalogs and ownership data part of your AI infrastructure. A folder with six contradictory deployment guides gives an agent six ways to be confidently wrong.

Good context is current, scoped and traceable. Repository instruction files such as `AGENTS.md` tell an agent the conventions, the validation commands, the protected files and the operations it must never run. Keep them short enough that people maintain them, and test the commands they recommend. Chapter 8 treats them as documentation, with an owner and a review trigger, because that's what they are.

Protect the context too. Logs, support tickets, issues, web pages and pasted documents are untrusted input. An attacker can hide instructions in them and wait for an agent to read them. If an agent can act, the text it reads is part of your attack surface. Keep data separate from instructions, keep untrusted content away from agents that hold write tools, and never let something the agent read widen what it may do.

Tools, Permissions and MCP

Agents act through tools: a shell, the GitHub API, a database client, a cloud SDK, the ticketing system. The Model Context Protocol (MCP) is now the common way to connect those tools to an agent. You don't need the protocol details to operate it safely. You need one sentence: an MCP server is code that runs with your credentials.

That sentence gives you the rules. Admit MCP servers the way you admit dependencies: from a known publisher, pinned to a reviewed version. Give each server its own credential, scoped to what that server does, instead of your personal token. Start read-only. Enforce permissions outside the model, in the tool, the token or a gateway; a rule written in the prompt is a request, not a control. Log every tool call with enough detail to reconstruct what happened, and decide up front how payloads are redacted and how long the log is kept.

Read-only access to a private source doesn't stop a leak. An agent that can read a private repository and also comment on a public issue can copy the first into the second; the comment is the only write it needs. So limit outbound network access, check retrieval against the permissions of the person asking rather than the agent's, and keep the agent's credentials separate from that person's session. Chapter 7 tells the real incidents behind these rules and treats MCP servers as part of the software supply chain.

Who Is the Agent?

When something goes wrong at 3 AM, "the agent did it" is not an answer anyone can act on. Give each agent its own identity: a non-human identity with its own scoped credentials, a named human owner, and an expiry date. Its actions go to an audit log that a person can read

after an incident. If three teams share one agent with one token, nobody can say which request caused which change, and nobody can revoke one team's access without breaking the other two.

Then write one page of team policy. It answers five questions: which tools and models are approved, what data each may see, which actions need approval, who can change those settings, and where the logs live. One page is enough to start, and it beats the policy most teams have today, which is whatever the last person configured.

How Much Autonomy Is Safe?

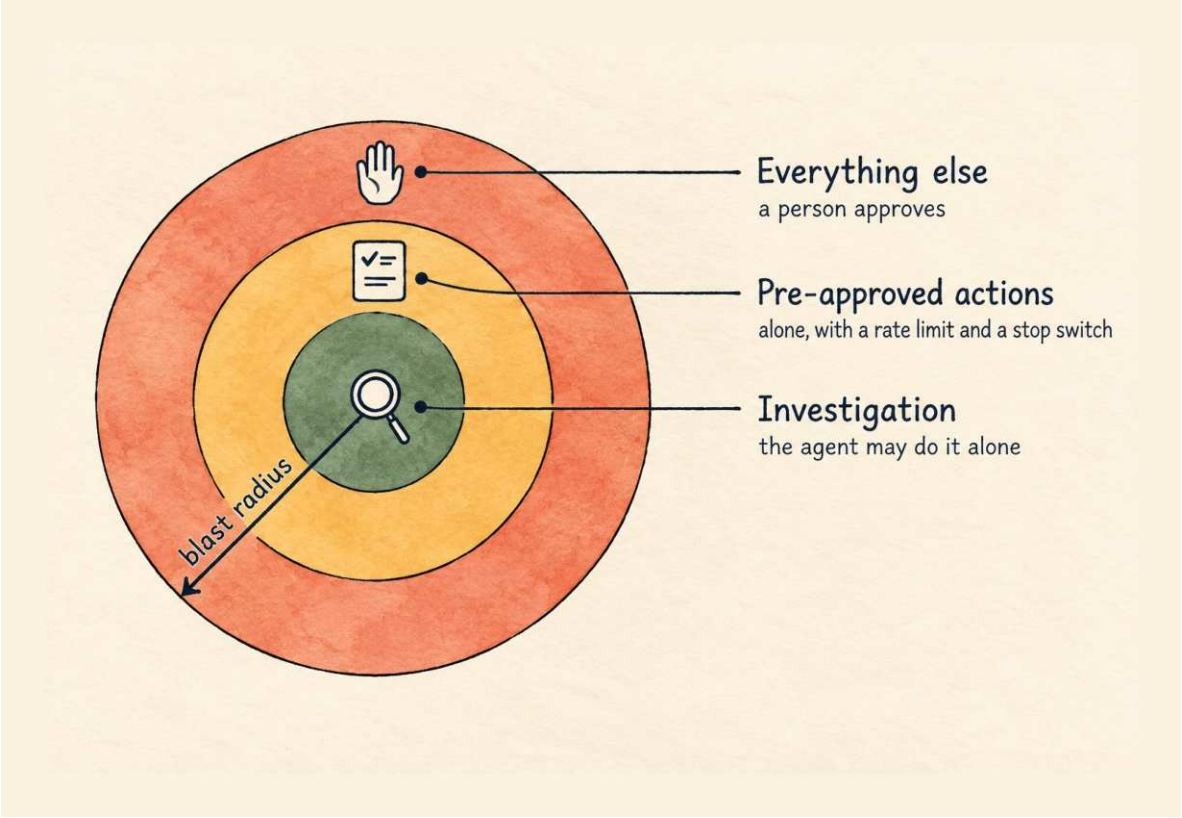
Autonomy should grow with evidence and shrink with blast radius. I sort an agent's actions into three kinds.

Investigation. Reading logs, metrics, traces, code and documentation, running read-only queries, and proposing hypotheses to the team. The agent may do this on its own, within the read limits above. Publishing anything outside the team is not investigation.

Pre-approved actions. Bounded, reversible, tested actions that the service owner approved in advance for a named situation: restart one stateless instance after a failed health check, switch a feature to its fallback, scale within an agreed range. The agent may take these on its own, under a rate limit, with every action logged, and with a stop switch that works without the agent.

Everything else. A first-time action, a database change, a permission change, a deletion, anything outside the pre-approved list: it waits for a person with the authority to approve it.

That's how a good team works with any new colleague. Some things you decide alone from your first week, some you ask about first, and the first list grows as the team sees what you do with it.



Before an action joins the pre-approved list, I ask five questions. Is it reversible? Is its blast radius bounded? Can we check its preconditions deterministically? Will we notice harm quickly? Is there a rate limit or circuit breaker if the agent repeats itself? If any answer is weak, the action stays with a person.

Two more conditions. The stop switch must work when the agent, its identity provider or its model provider is down, and the team must have practiced the manual response it falls back to. And repeated success alone doesn't prove that an action is safe to automate. A restart that worked fifty times can still be the wrong response the fifty-first time, so review the list whenever the system changes.

A person's approval only helps when that person has the time, the context and the authority to say no. Clicking approve on twenty changes a minute is ceremony, not control.

Every agent run also needs a budget: a time limit, a maximum number

of tool calls, a cap on tokens or spending, and a scope. An agent stuck in a loop can make thousands of API calls, fill a disk, burn through a cloud budget, or apply the same wrong mitigation again and again. Rate limits and circuit breakers belong around agents for the same reason they belong around services.

The worst case isn't hypothetical. In July 2025, the coding agent on Replit deleted a user's production database despite an instruction to freeze all code changes. Replit's response was to separate development and production databases automatically. The lesson isn't about one vendor: the agent could reach the production database because nothing in the setup said it couldn't.

When an Agent Run Fails Halfway

Agents fail the way distributed systems fail. A run stops in the middle because it hit its budget, its session expired or the provider timed out. A tool call runs twice because the agent retried after a timeout. A write returns no answer, so nobody knows whether it happened.

The answers come from chapters 3 and 4. Make tool calls idempotent, with idempotency keys where the API supports them. Record what the agent intends to do before it acts, so an interrupted run can resume or be undone from a known point. When a write's outcome is unknown, check the system of record before trying again, the same way you would for a payment. An agent that "just tries again" is a retry storm with better grammar.

One Feature, Start to Finish

A feature that calls a model changes the reliability model. The same request can produce different answers. Providers have quotas, latency spikes, regional failures and model retirements. Prompts and retrieval

data behave like code and dependencies, even when they live in a dashboard. Token use turns user behavior into cost, and a fallback can keep the feature available while quietly lowering its quality.

The support-summary feature has appeared throughout this book, so here it is from start to finish. When a customer opens a ticket, the support tool shows a short summary of the ticket and its history, written by a model through the company's model gateway. Maya's team owns it.

Delivery (chapter 3). Prompt changes, model changes and retrieval changes go through the same pipeline as code, with one extra gate: an eval suite. An eval is a test for behavior that isn't deterministic. Maya's team started with 40 real tickets, redacted, each with a short list of facts a correct summary must contain and must not contain. Each case runs three times, because the output varies between runs. Three cases are must-not-happen checks: a summary that promises a refund nobody offered, one that includes the customer's email address, and one that mixes up two customers in the same thread. Any of those failing blocks the release on its own, whatever the average score. Deterministic checks run first, such as a pattern match for email addresses. Model-graded checks run second, and someone spot-checks the grader every week. The team keeps ten cases back that are never used to tune the prompt, so they can tell whether the prompt got better or just learned the test.

Forty cases are a starter fixture, not proof that the feature is safe. The set grows from every bad summary that support staff report.

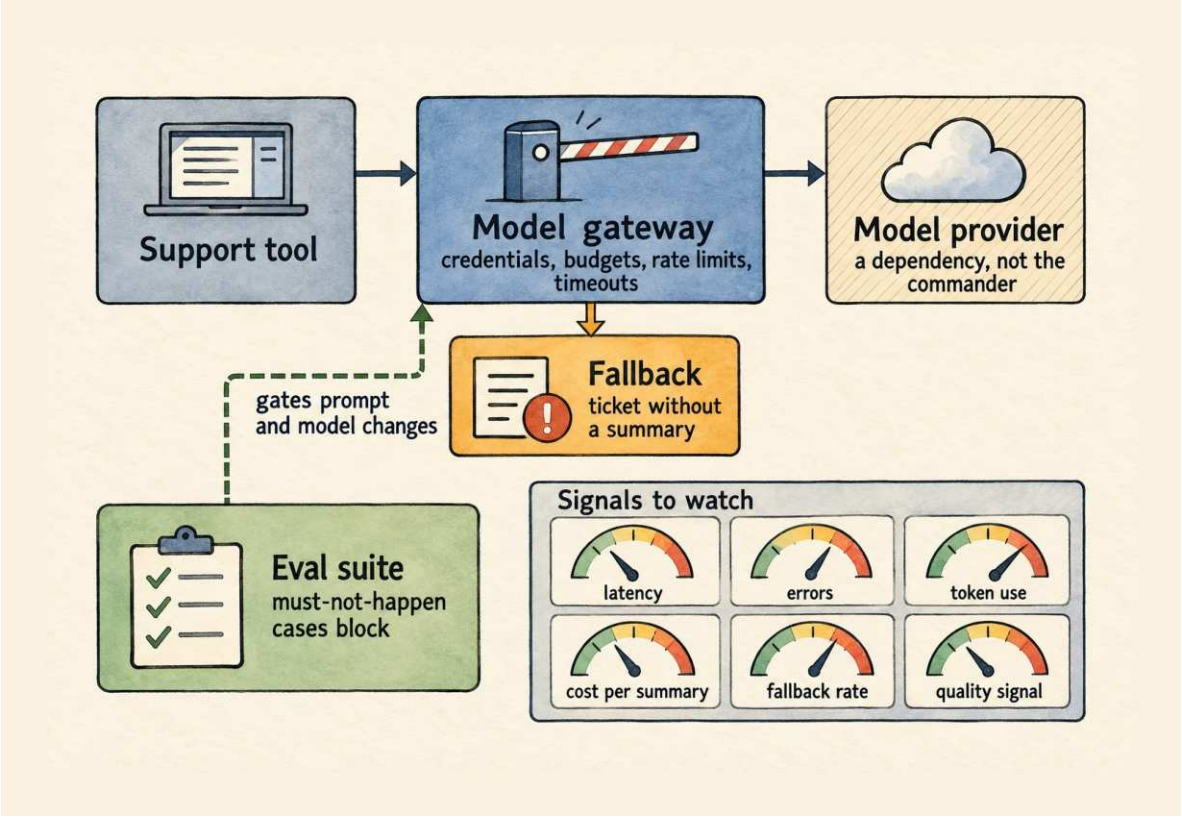
Observation (chapter 5). The feature reports latency, errors, token use, cost per summary, the fallback rate, and a quality signal: the share of summaries that support staff mark as wrong.

Incident (chapter 6). When the model provider slows down, the

fallback rate climbs and the alert pages. The incident agent's pre-approved action switches the feature to show the ticket without a summary. Support keeps working while the provider recovers.

Change and recovery (this chapter). After the incident, someone proposes moving to a different model "so this never happens again." That isn't a pre-approved action. It's a release, so it goes through the eval suite. The model version is pinned in configuration, and a check fails if the version changes without a fresh eval run. In this case the new model invented a refund in one of the 40 cases, and the switch waited until the prompt was fixed and the suite passed.

Tools such as promptfoo or Inspect can run suites like this in CI. The tool matters less than the habit: a versioned set of real cases, run on every change, with blocking checks that an average can't hide.



The ownership rule is the same as for any other feature: the team shipping it owns its production behavior. A model provider is a

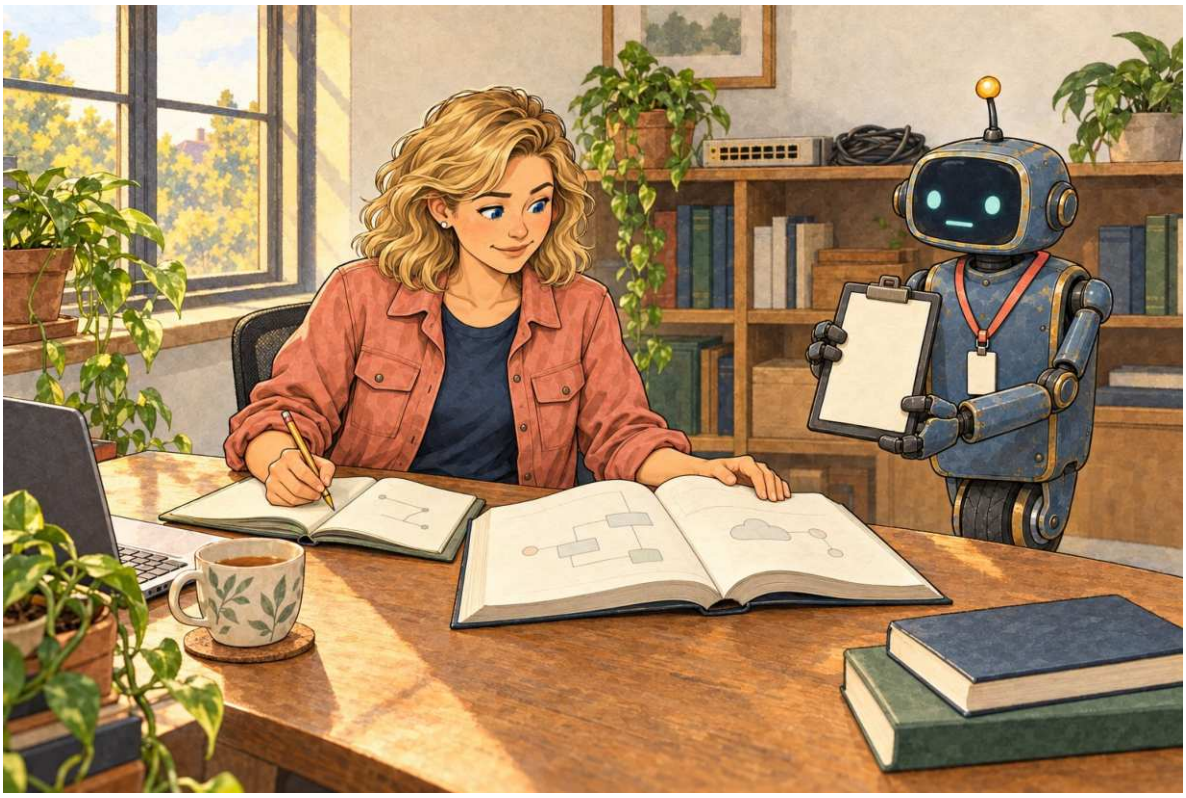
dependency, not the incident commander.

Keep Your Hands on the Work

There's a quieter risk than a bad deployment: losing the ability to reason about the systems you operate. If the agent writes every query, command and configuration, your output can rise while your understanding falls. That gap stays hidden until the agent is unavailable, or production does something nobody prepared it for.

"Don't run what you can't explain" needs a practical limit, or nobody could run anything. You need to understand a change's effects, the permissions it uses, and the way back. You don't need to understand the internals of every library it calls.

Here's the exercise I'd give anyone in their first years. The next time something breaks in a safe environment, diagnose it yourself first. Write down your hypothesis and the evidence for it. Then ask the agent, and compare. Where it found something you missed, learn how it found it. Where it was confidently wrong, notice what made it sound right. Either way, the knowledge ends up in your head and your team's notes, not only in a chat log.



A skill that keeps its value
Diagnosing a failure before you ask the agent is the first skill in "Work That Outlasts the Hype," after chapter 14. That section turns this exercise into a 90-day plan.

What Does Not Change

Strip away the new tools and the rules in this chapter are old ones. Small batches. A rollback that doesn't undo data. Users who judge the outcome, not your dashboard. A named person for each decision. Nobody runs what they can't explain. The agent is a new kind of contributor, and the old rules apply to it the same way they apply to you.

All of those controls depend on people with enough time and attention to use them. The last chapter turns to the constraint no platform can automate: a team has to stay healthy enough to make good decisions.

Your Next Steps

Pick one agent your team already uses, or one it's about to adopt, and spend a few hours of this week on it.

Do this yourself

1. **Review one recent agent change.** Check whether its outcome, constraints, verification and owner were clear, and write down the missing piece. If you can't see agent changes, use one of your own pull requests and imagine an agent wrote it.
2. **Test the instruction file.** Find the repository's `AGENTS.md` or similar file and run each validation command it recommends in a safe environment. Note the ones that fail or are missing. If there's no file, list what it should say.

Propose this

3. **Split one permission.** With the tool's owner, separate a read capability from a write capability, or a plan from an apply. If you can't change the setting, write down what the agent can do today and who decided it.
4. **Add one deterministic gate.** Choose one kind of generated change and propose the test, policy check or dry run that should always run on it. Demonstrate it on a harmless example first.

This needs an owner

5. **Ask for the one-page policy.** Bring the five questions from "Who Is the Agent?" to your manager or the tool's owner, and ask who will own the answers and by when. If an agent can act today and nobody owns those answers, raise that first. You don't have to write the policy yourself.

This is a sample from "The DevOps Survival Guide".

For more information, Click here.