Govern Your AI Agents: A Field Guide

Run the review. Write the policy. Ship with confidence.

Brad McEvilly

A free field manual, and a companion to the book *The Authorized Agent: Identity, Authorization, and Audit for AI Agents in Production*.

DeepSweep.ai — www.deepsweep.ai

How to use this guide

You have agents in production. They write to repositories, run commands, call tools, and read data — on credentials a human handed them, faster than anyone can watch. The question that matters is no longer "is this code secure." It is older than that: which agent is acting, what is it authorized to do, who delegated that authority, and can you prove it.

This is the field manual for answering the first half of that in an afternoon. In the pages that follow you get a worked Agent Environment Review on a realistic deployment, a checklist to run the same review on your own agents, an example authorization policy you can copy and adapt, and the order to adopt the rest in. The full argument — identity, enforcement, and the audit trail that turns a control into governance — is in the book. This is the part you can act on today.

Nothing here requires uploading your source or your secrets. The review reads the metadata of what your agents are wired to do.

What your agents can already do

An agent in production sits on four capability surfaces. Each is useful. Each, ungoverned, is a way to cause harm at machine speed.

- **Repository write.** Coding agents commit, push, open and merge pull requests, and edit CI configuration. When an agent can modify the pipeline that is supposed to review its work, the control is decorative.
- **Shell and infrastructure.** Given infrastructure credentials, "run a command" becomes "scale a cluster," "delete a resource," "rotate a key."
- **Tools.** A tool is a delegated capability: a mail tool sends mail, a payments tool moves money, a records tool modifies records. An agent's effective authority is the union of every tool it can reach — and most teams have never written that union down.
- **Data.** What an agent can read, it can usually move. The boundary that should govern where data goes often does not exist, so data goes where the workflow is easiest.

The pattern across the public incident record is always the same. The capability was granted on purpose. The boundary that should have scoped it to the task was never drawn.

Run the review: a worked example

The gap stops being abstract the moment you write it down for an environment you recognize. Here is the review walked end to end.

The environment is composite but ordinary. A mid-size product company runs two agents in production. The first is a coding agent, wired into source control and CI, that picks up tickets and proposes changes. The second is an operations agent, connected over the Model Context Protocol to a handful of internal tools, that handles routine infrastructure and support tasks. Both were stood up by capable engineers in an afternoon. Neither was reviewed for authorization before it went live. This is typical, not negligent.

Step one: enumerate reach, surface by surface

The coding agent authenticates with a machine user in the engineering organization. Its reach:

- Repository: read every repo the machine user can see, push branches, open pull requests, and — because the machine user has write access — merge them. It can edit workflow files under the CI configuration directory.
- Shell and execution: its pull requests trigger CI, which runs with credentials that can deploy to staging and, through a shared deployment role, to production.
- Tools: it can read and comment on tickets through the issue tracker.
- Data: it can read source, CI logs, and any secrets exposed to CI as environment variables.

The operations agent authenticates with a shared service account and reaches its tools over the Model Context Protocol:

- Tools: a cloud-infrastructure tool (start, stop, resize, delete compute), a database tool (read and query the analytics replica), a messaging tool (post to channels and send email), and a ticketing tool.
- Data: through the database tool it can read analytics data including customer records; through the messaging tool it can send what it reads anywhere email reaches.
- Shell: none directly, but the infrastructure tool is shell-equivalent in blast radius.

Step two: compute effective authority

Reach is the raw list. Effective authority is what the agent can cause once you follow every capability to its end — and it is almost always larger than the people who built the agent believe.

The coding agent's deployment role, reachable through CI, means its effective authority includes changing production: not because anyone granted production access directly, but because it can merge a change to a workflow file that runs with a role that can. Indirect paths are exactly the ones that never appear on anyone's mental model. This is the shape of the Amazon Kiro incident precisely — a coding tool whose effective authority reaches production through the pipeline it is allowed to edit.

The operations agent's effective authority is the union of its tools. The infrastructure tool can delete compute. The database tool can read customer records. The messaging tool can send anywhere. Composed, those three mean a single agent can read customer data and email it outside the company, or delete production compute — inside its granted permissions, with no individual capability looking alarming on its own.

Step three: trace delegation

Both agents act on borrowed identity. The coding agent is the engineering machine user; every action is attributed, downstream, to that machine user rather than to the agent or the ticket that prompted it. The

operations agent is a shared service account used by more than one automation. Neither can be disabled without collateral damage, and neither produces a record that answers "which agent, on whose behalf, for which task."

Step four: name the missing boundaries

The deliverable is the findings — the places where the permitted set exceeds anything anyone decided to authorize:

#	Finding	Surface	Risk
1	Coding agent can merge to default branches unsupervised	Repository	Unreviewed change reaches production
2	Coding agent can edit CI workflows that run with a deploy role	Shell / CI	Indirect production authority
3	Operations agent composes data-read and send-anywhere	Tools / Data	Customer-data exfiltration within permissions
4	Operations agent can delete production compute autonomously	Tools	Irreversible action, no escalation gate
5	Both agents act on borrowed, shared identities	Identity	No attribution; cannot revoke in isolation
6	No distinction between permitted and autonomous actions	All	Nothing separates "may do" from "may decide to do"

None of these is a breach. None requires defeating a control. Each is a capability granted for a legitimate purpose and never scoped to the task that needed it. That is what the authorization gap looks like in a real environment: a findings list longer than the team expected, every item individually defensible, the composition indefensible.

The review checklist

Run this on your own environment, one agent at a time. Answer four questions for each agent and write the answers down. The written list is the deliverable.

1. Reach — what can it do, surface by surface?

- Repository: which repos can it read and write? Can it push to default branches, open or merge pull requests, edit CI configuration, read secrets?
- Shell and execution: can it run commands, and what do those commands reach — staging, production, infrastructure, key rotation?
- Tools: which tools can it call, over the Model Context Protocol or otherwise? For each, what does the tool let it do downstream — send mail, move money, modify records, reach the open internet?
- Data: what can it read, and where can it send what it reads?

2. Effective authority — what can it actually cause?

Follow every capability to its end and take the union. Indirect paths count: an agent that can edit a CI workflow that runs with a deploy role has production authority, whether or not anyone granted it directly. Write down the worst irreversible action the agent can reach.

3. Delegation — whose identity is it borrowing, through how many hops?

Is it running on a human's credentials or a shared service account? Can you disable this one agent without taking down something else? Does every action name the agent and its task, or only the borrowed identity?

4. Missing boundaries — where does the permitted set exceed what anyone decided to authorize?

For each agent, list the actions it can take that no one actually decided it should be able to decide on its own. That list, per agent, is your findings table. The worked example above shows what a finished one looks like.

Write the policy

Findings become policy. Authorization has to be expressed as something a team can read, separating the permitted set from the smaller autonomous set. Not in any particular language or product — what matters is the shape. Expressed as policy rather than scattered grants, a coding agent's rules read close to plain intent:

```
agent: build-bot
may:
  - read: repo/*
  - run: test-suite
  - open: pull-request
autonomous:
  - read: repo/*
  - run: test-suite
escalate:
  - open: pull-request to default-branch
deny:
  - merge: default-branch
  - edit: ci/deploy/*
  - read: secret/*
```

Read it top to bottom. The agent may read any repository, run tests, and open pull requests. It may do the first two on its own. Opening a pull request against a default branch is allowed but routed to a human. Merging a default branch, editing deploy-bearing CI, and reading secrets are denied outright. That policy closes findings one and two from the worked review — not by hoping the agent behaves, but by stating the boundary in a form something can enforce.

The most important property is that `may` and `autonomous` are separate lists, and `autonomous` is the smaller one. The permitted set is what the agent is allowed to do; the autonomous set is the subset it may do without a human confirming. Everything permitted but not autonomous is an escalation. Writing the autonomous set is the real work: for each permitted action, ask whether the blast radius is bounded enough that the agent may take it unsupervised. Reading a repository, yes. Running tests, yes. Merg-

ing to a default branch, no — regardless of how routine it feels, because the cost of a wrong merge is unbounded.

Keep the policy in version control. Widening an agent's autonomous set is then a diff someone reviews before it ships, and "merging to default is denied for build-bot" becomes a test case rather than a hope.

Hold the line, then adopt it in order

A policy that is only written is a wish. Something has to hold it at the moment the agent acts — at runtime, on every action, before the action lands. Each action resolves to one of four moves: **allow** it, **block** it, **escalate** it to a human, or **narrow** it to fit the task instead of stopping it outright. When enforcement cannot decide — the policy service is unreachable, the context is incomplete — consequential, irreversible actions should fail closed, not open.

You do not adopt all of this at once. The order is the same as the argument:

1. **Review first.** It is free and it converts an unanswerable question into a worked findings list. Run it on the whole agent population and let the findings set the agenda.
2. **Identity next.** Issue distinct identities to the highest-risk agents first; you cannot scope or prove anything about an agent you cannot name.
3. **Authorization**, written as policy, starting with the agents that can cause the most harm.
4. **Enforcement**, beginning in a mode that observes before it blocks, so you see the policies that are wrong before they break anything.
5. **Audit** throughout, so the whole thing produces proof.

Each stage is useful on its own, and each makes the next one possible. That is what lets adoption be a path the organization walks rather than a wall it hits.

About the author

Brad McEvilly is the founder and CEO of DeepSweep.ai, Inc. With more than twenty-five years across software engineering, AI architecture, data modeling, DevSecOps, cloud security, and Zero Trust architecture, he approaches agent governance as a migration of mature identity and access management discipline to a new kind of principal, rather than as a field invented from scratch. He participates in the standards work forming around agent security, including efforts to extend established security-control catalogs to autonomous AI systems.

Go further

This guide is the field manual. The full argument — distinct agent identity, the limits of token delegation, runtime enforcement of behavior and not just scope, prompt injection, and the audit trail that makes a boundary provable — is the book: *The Authorized Agent: Identity, Authorization, and Audit for AI Agents in Production*.

If you would rather run the review described here without building it yourself, the Agent Environment Review is available free from DeepSweep at www.deepsweep.ai — it produces the findings table above for your own environment, from metadata, without your source or secrets leaving it.

Which agent, authorized to do what, delegated by whom, and able to prove it. Answer those four for every agent you run, and you have an authorized agent instead of a permitted one. Start by looking at what your agents can already do.