# **Governing Multi-User AI Agents:** A CISO Reference for Per-Action Authorization

How to enforce policy on every agent action.

Arcade

# Executive Summary

An AI agent that acts in production systems on behalf of many users needs deterministic control on every action it takes. That control has to sit outside the model, at the action layer, because anything the model can reach, the model can be talked around. Agents should act as delegates of a named user, never as principals of their own, and every action should be authorized at the intersection of the user's permissions and the agent's scope.

## Key risks

- Agents given their own service accounts hold standing, over-scoped access that is tied to no person.
- Agents that inherit a user's access let a single prompt injection reach everything that user can touch.
- Guardrails inside the model are probabilistic. The same request can pass once and fail the next time, which is not a control you can test or show an auditor.
- Each agent logs on its own or not at all, so answering for one action means pulling logs from many systems.
- Reviewing every agent by hand makes security the bottleneck, and teams route around the wait.

## Recommended actions

1. Authorize every action outside the model, against your identity provider, at the intersection of user permissions and agent scope.
2. Authorize just-in-time for each action, and keep credentials vaulted so the agent never holds them.
3. Put every agent, tool, and client on one control plane, with one record per action that the agent cannot alter, streamed to your SIEM.
4. Review the runtime once rather than each agent, so every new agent inherits your authorization model, policies, and audit trail.
5. Keep a human behind actions that are hard to undo. Per-action authorization stops privilege escalation. It does not stop an injected instruction that stays inside what the user is already allowed to do.

Arcade

## Key Terms

**Agent.** Software that uses a model to decide on and carry out actions in other systems, such as creating a ticket or reading an account record.

**Action layer.** The point where an agent's request becomes a call to a tool or a system. Governance is enforced here, on the single path every action has to take.

**Guardrail.** A probabilistic control inside or around the model: system prompts, content filters, and model-judged checks. Guardrails reduce bad outputs. They do not decide whether an action is allowed.

**Governance.** Deterministic control outside the model, at the action layer. It decides whether an action may happen at all, and a denied action never reaches the target system. The same input produces the same decision on every run.

**Principal and delegate.** A principal holds an identity and permissions of its own. A delegate acts with authority borrowed from a user and holds none of its own. In this reference, agents are delegates, never principals.

**Intersection (of user and agent).** The set of actions allowed by both the user's permissions and the agent's scope. The agent's scope sets the ceiling, and the user's permissions set the boundary.

**Policy enforcement point.** The component that applies an authorization decision to each action. Policy decisions stay in your identity provider, DLP, and policy engine. The enforcement point is where they take effect.

**Runtime.** The layer where agent actions are authorized, executed, and recorded. Reviewing the runtime once covers every agent that runs on it.

**Control plane.** The single place where you grant, revoke, and version the tools each team and agent can use, and where you see what agents exist, who owns them, and what they have done. The control plane manages the runtime.

**Just-in-time (JIT) privileges.** An agent asks to do something, and the authorization decision is made for that one action. The action receives exactly the privileges it needs, and the agent holds none afterward. The credential is vaulted apart from the agent and the model, and the runtime manages its lifecycle, including refresh and rotation. JIT removes standing over-privilege and bounds the blast radius. It does not cap the cumulative effect of many individually authorized actions.

**Standing access.** Privileges that are granted once and held indefinitely, whether or not they are in use.

**Blast radius.** The set of systems and data an agent can affect if it is compromised or misused.

**Prompt injection.** A hostile instruction hidden in content the model reads, such as a document or an email, which the model cannot reliably tell apart from a legitimate one.

**Confused deputy.** An agent tricked by an injected instruction into misusing access it was legitimately trusted with.

## Why agent access models leave security exposed

*Giving an agent its own service account or the user's full access are the two common patterns, and neither can be reviewed at scale.*

To do real work, an agent needs access to production systems and the data inside them. Two patterns have taken hold for granting it, and both leave you exposed.

In the first, the agent gets its own service account: standing, over-scoped, and tied to no person. In the second, it inherits the user's access, so one prompt injection can reach everything that user can touch.

Neither can be reviewed at scale, so security reviews each agent by hand, becomes the bottleneck, and still owns the gap when an agent takes a destructive action. Governance means enabling access while enforcing deterministic control on every action the agent takes.

**This reference follows three goals:**

- **Enforce:** Authorize every action at the intersection of the user's permissions and the agent's scope, with deterministic controls outside the model, not model-level guardrails.
- **Govern:** Keep one named, per-action record of what every agent did and on whose behalf.
- **Enable:** Say yes to the next agent without reviewing each one by hand.

“Giving an agent its own service account or the user's full access are the two common patterns, and neither can be reviewed at scale.”

Arcade

# 1. Why guardrails can't gate production access: probabilistic vs. deterministic control

*Guardrails reduce bad outputs. Governance stops bad actions. Only a control the model cannot reach counts as governance.*

Most agent safety today lives inside the model: system prompts, content filters, and model-judged checks. These are guardrails, and they are probabilistic. They usually work, until a jailbreak or an unusual prompt gets through. A control the model can reach is a control the model can reason around, so the same request can pass one time and fail the next. That is fine for steering tone in a chatbot. It is not for a control you can test, attest to, or put in front of an auditor, and for an agent touching money or data a rare failure is an incident.

Governance is different. It lives outside the model, at the action layer, on the single path every action has to take. It decides whether an action is allowed to happen at all, and a denied action never reaches the system.

The test is reach: if the model can reach the control, the model can route around it, and a control the model can route around is not deterministic. A control that is not deterministic is not governance. A runtime can still run probabilistic checks, like anomaly scoring or model-judged review, and those have their place, but they sit alongside governance and can't replace it. The controls that gate an agent's access to production data have to be enforced deterministically, on rules that do not depend on what the model decided or what a prompt injection told it, so the same input yields the same decision on every run and you can prove it.

Guardrails reduce bad outputs. Governance stops bad actions. You need both, in different layers.

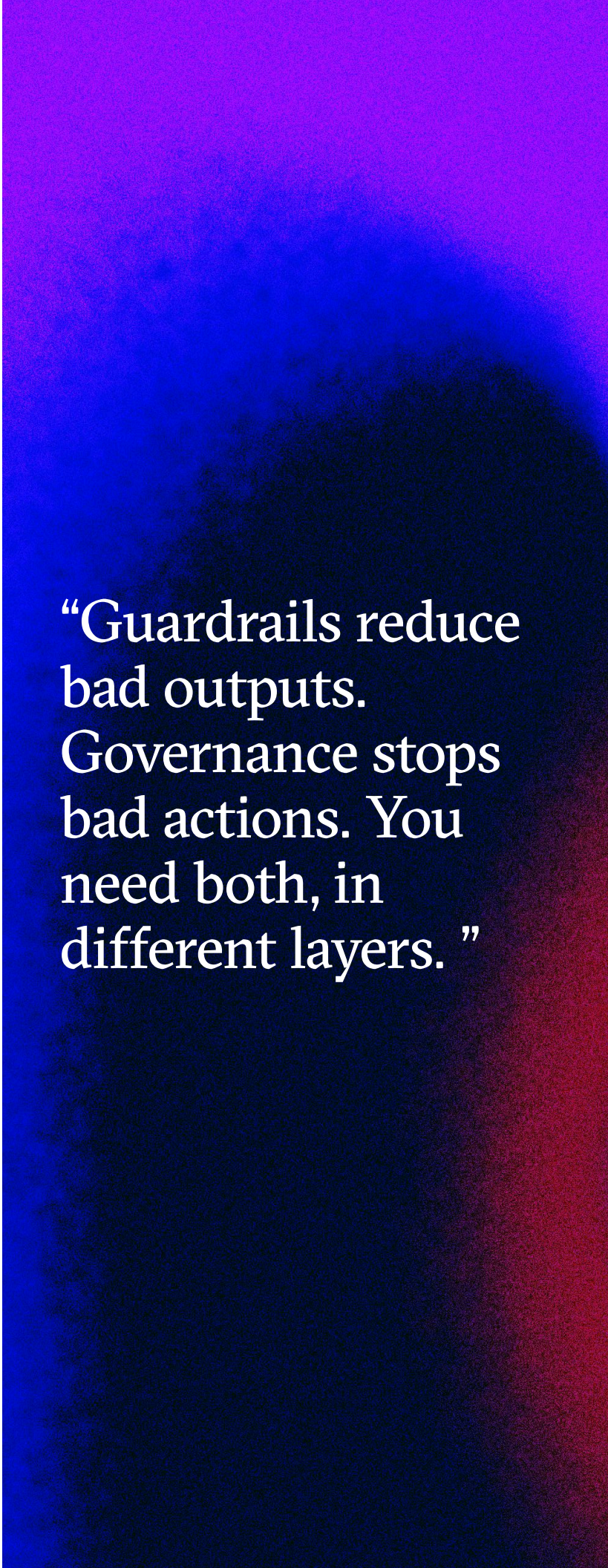
| Dimension | Probabilistic guardrails | Deterministic governance |
|---|---|---|
| Where it runs | Inside or around the model (system prompts, filters, model-judged checks) | Outside the model, at the tool call |
| How it decides | Predicts whether the output looks safe | Evaluates the action against identity and policy |
| Failure mode | A jailbreak or novel prompt slips through; the same prompt can pass one day and fail the next | The action is denied unless explicitly allowed; a blocked call never reaches the system |
| Auditability | You cannot audit a probability | Same input, same decision, on every run |
| What it protects | The conversation and the output | The action and the system it touches |
| Best for | Reducing bad outputs, steering tone | Stopping unauthorized or destructive actions |

Read the table above by where each control runs. Guardrails sit inside or around the model, so a jailbreak or an unusual prompt can get past them, and the same request can pass one day and fail the next. Governance sits at the tool call, outside the model.

An action is denied unless it is explicitly allowed, and a blocked call never reaches the system. That is why only governance gives you the same input, same decision result on every run, which is what you can test and show an auditor. The two work in different layers, and you need both.

## Key enforcement points for deterministic controls

- Agent harness and client: which connectors, gateways, and tools an agent is allowed to use at all
- Foundation and configuration: team scoping, allow and deny policy, and least-privilege scope definitions, set before any action runs
- Tool access: which tools a user's agent can see, at the intersection of the user's and the agent's permissions
- Pre-execution gate: per-action authorization, input validation, rate and spend limits, and credential injection
- Execution and in-flight: sandboxing, egress and outbound DLP, and idempotency
- Post-execution: output validation, and DLP and redaction before results reach the model
- Always-on: an immutable audit record at every point, and a kill switch that can stop an agent at any of them




Arcade

# 2. Why governing agents is different from governing users

*Agents act at machine speed for many users at once and can be manipulated by text, so human-oriented access controls need to be applied per action, outside the model.*

Agent governance looks like identity and access management, so teams reach for the playbook they use for people. It does not fit, for the five reasons below.

- **Scale and speed.** A person takes one action at a time. An agent can take thousands in the same span, so a small error rate turns into a large number of bad actions fast.
- **Many users, one actor.** A single agent acts for many people. Who did this is never just the agent. It is the agent, on behalf of a specific user, and your controls and your logs have to carry both.
- **No accountability.** A person can be trained, warned, and held accountable. A model cannot, so safe behavior has to be enforced from outside the model.
- **Prompt injection.** A model acts on whatever text it reads and cannot tell a legitimate instruction from a hostile one hidden in a document or an email. A prompt injection turns the agent into a confused deputy, tricked into misusing the access it was trusted with.
- **Unreliable testing.** Passing QA tests can’t prove the agent won’t take destructive actions. Assurance has to come from a control on each action, not from testing.

The right model for agent identity is delegation: the agent acts as a delegate of the user, never as a principal of its own, and every action is authorized at the intersection of the user's permissions and the agent's scope. The agent's scope sets the ceiling, the user's permissions set the boundary, and the credential never reaches the model.

None of this replaces the controls you already run. Your identity provider, your data classifications, and your DLP and access policies stay the source of truth. Agent governance layers on top, extending them with the agent's identity and scope, rather than recreating them.

# 3. Governance fundamentals for multi-user agents: enforce, govern, enable

*Three questions cover agent governance: can this agent take this action for this user right now, what did it do and for whom, and how do we approve the next agent without a manual review?*

Three questions cover agent governance.

- **Enforce:** can this agent, on behalf of this user, take this action, right now?
- **Govern:** what did this agent do, and on whose behalf?
- **Enable:** how do we say yes to the next agent without reviewing each one by hand?

The practice areas below answer each one, naming what to require, what evidence has to stay visible, and where human judgment still enters.

## Enforce: control every action

- **Authorize at the intersection of user and agent.** Resolve the user against your IdP on every action. Evaluate what that user may do and what the agent is scoped to do, and allow only their intersection. This is the policy enforcement point for agent actions, and it sits outside the model.
- **Grant privileges just-in-time.** An agent asks to do something, receives exactly the privileges that one action needs, and holds no standing access afterward. This is least privilege applied per action. It bounds the blast radius of a compromised agent. It does not erase it, because an agent can still iterate within what it is allowed, so treat just-in-time as removing standing over-privilege, not as capping cumulative effect.
- **Keep credentials out of the model.** Vault tokens apart from the agent and the model, and manage their lifecycle, including refresh and rotation, in the runtime. The agent never holds the credential, so exposure of the model context does not expose your access.
- **Scope tool access per user.** Show an agent only the tools its user is allowed to invoke. An agent that can't see a tool can't call it and can't be tricked into misusing it.
- **Inspect and block at the action boundary.** Deny or modify inputs before an action runs, and inspect, redact, or block output after it returns. Wire your existing DLP and policy checks in here, and fail closed when a check cannot complete.
- **Deny self-escalation.** An agent cannot widen its own scope beyond a pre-authorized maximum, even under an injected instruction.

**What per-action authorization does and does not stop.** It stops privilege escalation. It does not stop prompt injection. An injected instruction that stays inside what the user is already permitted to do is an authorized action, and no enforcement point catches that on permissions alone. This is the confused-deputy problem. It is why the per-action record in Section 6 matters, and why a human stays behind consequential actions.

## Govern: know what every agent did

Boards and auditors already treat what agents do as security's responsibility. The pain shows up in forensics. A CISO running several hundred agents in production called the investigation after an unexpected agent action brutal when we spoke to him. "We've had to kind of pull together logs from multiple systems to try to figure out what happened."

- **Bring every agent under one control plane.** Teams build on different models, frameworks, and clients. Put them on one runtime so you have a single view of what exists, who owns it, what it can reach, and what it has done.

- **Record every action in a log the agent cannot alter.** Write one authorized record per action, attributed to a named user, a named agent, and a named system, kept outside the agent's reach, and streamed to your SIEM. When something goes wrong, you answer from one log with one query, not a manual pull across a dozen systems.
- **Leave no unlogged path.** The record is written by the enforcement point, not the agent, so the agent cannot skip it, suppress it, or slip an action through a client or connector that does not log. Route every client and connector through the control plane and close off direct access, and an action that isn't recorded can't run.
- **Grant, revoke, and version centrally.** Manage in one place the tools each team and agent can use, push a fix without tracking down where an agent runs, and revoke access when a connection should end.

## Enable: securely scale productive agents

- **Make the governed path the fastest path.** Review the runtime once instead of every agent that runs on it. Every team that builds on it inherits your authorization model, your policies, and your audit trail by default, across any model, framework, or client. The controls come with the runtime, so builders move fast and nothing routes around you.
- **Register every agent and tool.** Keep one registry of record for every agent, MCP server, connector, tool, and skill or instruction file in use, each with a named owner, a version, and its provenance. Nothing reaches production until it is registered, so there are no shadow agents or tools, and every team knows what is approved before they ship.

| Control | Agents without a runtime | Agents on a governed runtime |
|---|---|---|
| **Access model** | Holds a service account of its own, or inherits everything its user can reach. | Acts as a delegate. Every action is authorized at the intersection of user identity and agent scope, resolved against your IdP. |
| **Privileges** | Standing and over-scoped, tied to no person, granted once and held indefinitely. | Just-in-time per action. Credentials are vaulted apart from the model and the agent, with refresh and rotation handled for you. |
| **Enforcement** | Guardrails written into prompts, so the same request can pass once and fail the next. | Enforced outside the model. Your DLP, SIEM, and IdP policies fire on every action, with room to add your own. |
| **Audit** | Each agent logs on its own or not at all, so one question means pulling logs from a dozen systems. | One record per action naming the agent, the user, and the system, streamed to your SIEM. |
| **Change control** | No central way to grant or revoke tools, or to push a fix once an agent is running. | One control plane to grant, revoke, version, and apply changes across the org. |
| **Security review** | Every agent reviewed by hand. Reviews pile up and teams route around the wait. | Review the runtime once. Every agent built on it inherits your model, policies, and audit trail. |

The table above compares two paths. Without a runtime, each agent carries its own access model, its own logging, and its own security review, so risk and manual effort grow with every agent you add. On a governed runtime, authorization, credential handling, enforcement, audit, and change control belong to the runtime instead. You review the runtime once, and every agent built on it inherits the same model, policies, and audit trail. The practical result is that security stops being the bottleneck without giving up control of any single action.

# 4. The eight-step per-action authorization flow

Every governed action follows the same eight steps, whatever model or framework produced it, and every step is enforced outside the model.

1. **Request.** The agent requests an action on behalf of a named user.
2. **Resolve identity.** The user is resolved against your IdP.
3. **Evaluate.** The runtime checks policy at the intersection of the user's permissions and the agent's scope, plus your contextual and DLP rules.
4. **Decide.** Allow or deny. A denial stops the action and is recorded.
5. **Grant just-in-time.** The action gets exactly the privileges it needs. The credential stays vaulted and never reaches the model.
6. **Execute.** The action runs against the target system.
7. **Inspect the result.** The runtime validates the output and applies DLP and redaction before results reach the model. A failed check is blocked, not returned.
8. **Record.** The runtime writes one record naming the user, the agent, and the system, then streams it to your SIEM.

Four points in this flow carry the control. The decision is made before anything executes (steps 3 and 4), and a denial is recorded, so a blocked action leaves evidence. The credential is granted for the action and kept apart from the model (step 5), so exposing the model's context does not expose your access. The result is inspected before the model sees it (step 7), and a failed check is blocked rather than returned. And the record is written by the runtime, not the agent (step 8), so the agent cannot skip or suppress it.

# Eleven questions to ask before any AI agent reaches production

Each question comes with what a strong answer looks like and the evidence to require, which maps to what ISO 42001, NIST AI RMF, and NIST CSF 2.0 programs will ask you to show.

| Question to ask | What good looks like | Evidence to require |
|---|---|---|
| **Enforce · How does the agent get access to act in our systems?** | It acts as a delegate of the user, not as a principal of its own, with no standing service account. | The access model, and proof there is no standing over-scoped credential. |
| **Enforce · How is each action authorized?** | Every action is checked against the intersection of the user's permissions and the agent's scope, resolved against your IdP, outside the model. | An authorization decision log showing user, agent, action, and decision. |
| **Enforce · Are privileges scoped to the action or standing?** | Authorized just-in-time for the one action, and the agent never holds the credential. | The authorization decision for a sample action, and the lifetime and rotation policy for the vaulted credential. |
| **Enforce · Where do credentials live?** | Vaulted apart from the model and the agent. The agent never holds them. | Where tokens are stored, and how they are kept out of the prompt, model context, and logs. |
| **Enforce · Can the agent see tools its user may not use?** | No. Tool visibility is scoped per user. | The tool list an agent receives for a given user. |
| **Enforce · What runs before and after an action?** | Inputs can be denied or modified, output can be inspected or redacted, and checks fail closed. | The pre- and post-execution checks and their fail behavior. |
| **Govern · Can you show every action one agent took, and for whom?** | One record per action, attributed to user, agent, and system, in a log the agent cannot alter. | A single query that returns an agent's actions with attribution. |
| **Govern · Can the agent alter the logs of its own actions?** | One control plane, with versioning and central revoke. | The grant and revoke path, and how a fix reaches a running agent. |

Arcade

| Question to ask | What good looks like | Evidence to require |
| --- | --- | --- |
| **Govern · Can the agent act without a record being written at all?** | No. The audit record lives outside the agent's reach, so it cannot edit or delete the evidence of what it did. | Where audit logs are stored, who can write to them, and proof the agent has no path to modify them. |
| **Enable · How does a new agent inherit our controls?** | Not on the governed path. The enforcement point writes the record, not the agent, so no action through the runtime runs without one. Route every agent client and connector through the control plane, and close off direct access to the same systems, so the runtime is the only way in. | Which clients and connectors route through the control plane, how direct access to the same systems is closed off, and a sample record showing inputs, outputs, parameters, and timestamps. |
| **Enable · How does a new agent inherit our controls?** | It runs on a reviewed runtime and inherits the authorization model, policies, and audit trail by default. | The once-reviewed runtime, and what a new agent gets automatically. |

# A sample per-action authorization record

Read the record from the top. The decision says whether the action was allowed. The user, agent, and scope lines say who the action was for and what the agent was permitted to do. The action and system lines say what ran and where. The credential line shows the secret stayed vaulted and never reached the model. The correlation ID and sink let you pull this record, and every other record in the same chain, with one query in your SIEM.

```
ACTION AUTHORIZATION RECORD
decision          ALLOW
timestamp         2026-09-17T14:22:07Z
user              j.rivera@acme.com (resolved via Okta)
agent             wealth-assistant (scope: read:accounts, write:tickets)
action            crm.create_ticket
authorized_by     user + agent scope intersection matched; contextual policy OK
credential        vaulted; never exposed to the model
system            Salesforce (prod)
correlation_id    9f2c1d7a-4b02-4e77-9c3a-0d5e2f7a1a41
sink              SIEM (OpenTelemetry)
```

**Review notes.** This record shows the action was permitted and by whom. It does not certify the action was wise. An authorized action can still be the wrong one if an injected instruction stayed inside the user's permissions. Read a clean record as a starting point for review, not a verdict, and keep a human behind actions that are hard to undo.

# Your 60-second posture check

Run this on any agent you are about to approve. If you cannot check a box, that is where to look first.

- [ ] The agent acts as a delegate of a named user, not as a principal of its own
- [ ] No agent holds a standing, over-scoped service account, and none can widen its own scope
- [ ] Every action is authorized against the user and agent intersection, outside the model
- [ ] Privileges are granted just-in-time, and the agent never holds the credential
- [ ] Credentials are vaulted and never reach the model context or logs
- [ ] Each agent sees only the tools its user may invoke
- [ ] Inputs and outputs pass DLP and policy checks that fail closed
- [ ] High-risk actions require out-of-band human approval, routine ones do not
- [ ] Every action writes one record, attributed to user, agent, and system, in a log the agent cannot alter
- [ ] Activity streams to your SIEM in real time
- [ ] Tools can be granted, revoked, and versioned from one place
- [ ] A new agent inherits your controls by running on a reviewed runtime

Arcade

## Conclusion

Agents change the tempo of risk. The principles for controlling it still hold. Deleted, corrupted, or exposed data is the same damage whether a person or an agent caused it.

What changes is the path to it: an agent gets there faster, at machine scale, and without judgment, so how you prevent it has to change even though the goal has not. The controls you built for people – least privilege, a named actor behind every action, a record you can query – still apply. They hold only when they are deterministic and live at the action layer, where the model cannot reason around them. Assume the model will be jailbroken, and build so that a jailbreak can't escalate beyond what the user could already do. Govern the runtime once, authorize every action at the intersection of user and agent, keep the credential away from the model, keep a person behind the actions that matter, and record every action where the agent cannot touch it.

Done this way, security puts more agents into production and can answer for every action they take.

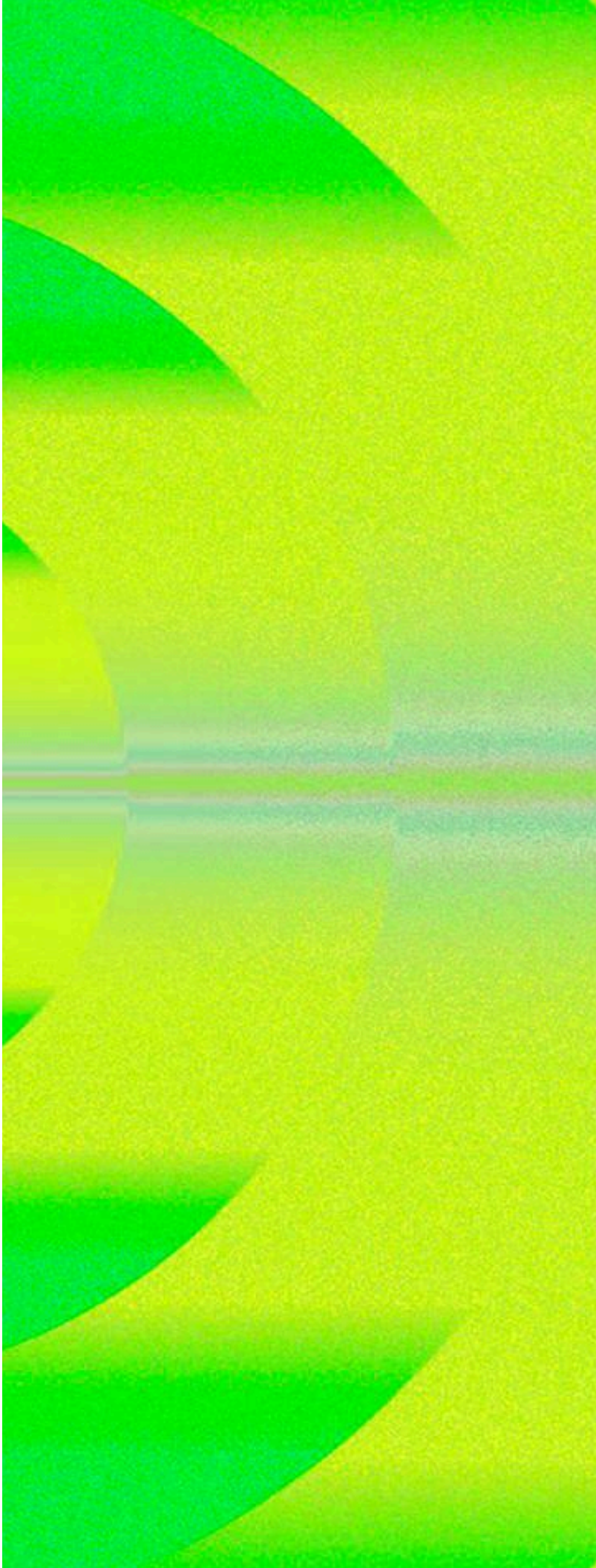
“Agents change the tempo of risk. The principles for controlling it still hold.”

## About Arcade.dev

Arcade is the actions runtime for governing enterprise AI agents. We enforce your security policies on every action, bring every agent and tool under a single control plane with full auditability, and clear the path to production for every agent. We grant privileges just-in-time, vault credentials separate from the agent, and hook into the IdP, DLP, and SIEM you already run.

**In production.** A top-five bank put a wealth-management agent into production on Arcade after security would not approve a custom auth build. One runtime review became the firm-wide standard. Dozens of production agents now serve 30,000 users across eight teams, connected to more than 50 systems, each inheriting the same authorization model, policies, and audit trail.

MORE ON AI AGENT GOVERNANCE



Arcade

# Get Started Today

Two paths to deploying production-ready agents with Arcade:

### REQUEST A DEMO & SEE IT IN ACTION

Our team will walk through your specific use case, show how Arcade's architecture solves your production barriers, and design a deployment plan for your agents. Arcade.dev

### SIGN UP & START BUILDING

Developers can start immediately with Arcade's free tier. Build your first agent with production tools, test multi-user auth flows, and validate the platform with your use cases before committing. Signup here

The difference between agents that demo well and agents that actually work comes down to infrastructure. MCP made the connection possible. Arcade makes it practical.

