Who Has Access? Humans, Accounts, AI Agents

2026-07-09

Kubernetes RBAC, OVH IAM v2, LiteLLM keys, and an AI coding agent compared on documentation, auditing, and technical enforcement of who has access. Part 3 of 6.

Part 1 split the technology-supply-chain question into two halves: what the platform is built on, and who — or what — actually has access to it. Part 2 answered the first half. This post answers the second, using systems already built and operated for this blog rather than new research — human access, service-account access, and (increasingly relevant) AI-agent access, all checked against the same bar: is it documented, is it audited, and is it technically enforced or just expected to be followed.

Human access

Kubernetes RBAC is the reference case for what technically-enforced human access looks like: a `Role` or `ClusterRole` grants specific verbs on specific resources, a `RoleBinding` attaches it to a user, group, or service account, and the API server rejects anything outside that grant — RBAC is purely additive, with no deny rules to misconfigure. The K8s/OpenShift Best Practices security post covers this in depth, including the one habit that undoes all of it: reaching for `cluster-admin` because a narrower role is more work to define.

OVH IAM v2 is a second, less mature example of the same idea for human users: policies attach to OVH users and grant specific actions (`objectstore:object:get`, `objectstore:object:put`, and so on), covered in the log-archiving security comparison. Its more interesting gap shows up one layer down, in how it handles machine identities — covered next.

Service-account access

The same K8s security post applies the human-access principle downward, to workloads rather than people: the default service account in every namespace gets no RBAC permissions at all, which is correct, and a workload that doesn't call the Kubernetes API needs none — only workloads that actually do should get a dedicated service account, scoped to exactly what they use, never shared across unrelated pods.

The harder question is how that Kubernetes-native identity gets federated to the cloud provider's own IAM, so a pod can reach object storage or another managed service without a long-lived secret sitting in a Kubernetes Secret. The log-archiving security comparison already documents this for AWS, GCP, and Azure — and checking STACKIT against the same bar fills in a gap that post didn't need to cover:

Provider	Workload identity federation	How it works
AWS	IRSA (IAM Roles for Service Accounts)	EKS's own OIDC provider issues short-lived credentials to a Kubernetes ServiceAccount bound to an IAM role — no static keys
GCP	Workload Identity Federation	A GKE ServiceAccount is bound to a GCP Service Account, which carries the IAM bindings
Azure	Azure Workload Identity (successor to AAD Pod Identity)	An AKS ServiceAccount is bound to a Managed Identity; the pod receives Azure AD tokens directly
OVH	None	No automatic federation between a Kubernetes ServiceAccount and an OVH IAM identity — credentials still have to be distributed by hand via the OVH API or CLI
STACKIT	No confirmed SKE-native equivalent	STACKIT's "Workload Identity Federation" runs in the opposite direction from the three rows above: a STACKIT service account can accept short-lived tokens from an *external* OIDC issuer — documented for CI systems such as GitHub Actions — to authenticate into STACKIT. That's the reverse of SKE's own cluster API server acting as an OIDC issuer that STACKIT's IAM trusts, which STACKIT's documentation does not describe at all; treat native SKE-to-IAM federation as an open question, not a confirmed gap or a confirmed feature

STACKIT sits in a middle position worth naming plainly: a real workload-identity-federation building block exists, unlike OVH — but it's built for external identities (CI/CD systems) authenticating into STACKIT, not for SKE's own pods authenticating outward to STACKIT's IAM the way IRSA, GCP's Workload Identity Federation, and Azure Workload Identity work for their respective Kubernetes services. Whether that pattern can be wired up to SKE another way is not something this post's sources confirm either way.

LiteLLM's per-user virtual keys, sitting in front of a shared self-hosted vLLM endpoint, are the same scoping idea outside Kubernetes entirely: instead of one shared bearer token for every client, each user or workload gets its own virtual key with its own budget limit, and every call is attributable to a specific key in the gateway's own logs — the audit trail a single shared API key can't give you.

AI-agent access

This is the newest, least settled piece of the access question, and arguably the most consequential one. An AI agent only becomes useful once it can act without a human approving every single step — which means the rights that make it useful are the same rights that make it dangerous. Granting

an agent broad execution access (API keys, shell access, cloud credentials, an inbox) hands over not just permissions but a share of actual control: the person who granted the access frequently cannot see, in real time or after the fact, what the agent did with it. Many AI-agent deployments in production today do not produce anything resembling a genuine audit log by default — not because it's technically infeasible, but because it simply isn't built in. Without one, "what did the agent just do" stops being a governance question and becomes a real, unanswerable gap. An agent with that kind of broad, unaudited access, misconfigured or misdirected, can shut down or delete production systems, or delete correspondence, before anyone downstream even notices — and if it happens, there is often no log to reconstruct what actually ran.

> ⚠ **Delegation isn**
>
> Everything above is about *access* — what an agent can technically reach and do. None of it changes who is *responsible* for what happens with that access. Granting an agent broad rights doesn't transfer accountability to the agent; the person or organization that granted the access, wrote the policy, and reviewed — or failed to review — its output still owns the outcome. "The agent did it" is not a liability disclaimer, no matter how autonomous the setup is.

An example of interactive, sovereign agent use

Claude Code, the AI coding agent used to help draft and maintain this blog, is one example of pushing back against that default — not a general solution to it, and not itself immune to the underlying risk. It works from the author's direction and critical review — deciding what to write, questioning and pushing back on drafts, and folding in the author's own operational experience — and operates under two overlapping controls that at least make its actions visible and interruptible.

The first is harness-level: **plan mode**, which forces the agent to lay out an approach and wait for explicit approval before touching any file, and the same permission-prompt mechanism that asks before running a shell command or tool call outside what's already been allowed. Every tool call is also recorded in the session transcript, giving a reviewable record of what the agent actually did — closer to an audit trail than most agent deployments get by default, though it is a session log a human has to read, not a centrally stored, tamper-evident trail the way Kubernetes audit logging or a LiteLLM key's call log are.

The second is written policy, not enforced by any API server: this very project's `CLAUDE.md` file has a "Git Safety Protocol" section stating plainly that the agent must never run `git commit`, `git push`, or `git add` on the user's behalf — not even mid-task, not even if asked. The agent is expected to follow that rule because it's written down and because the harness surfaces a permission prompt for the underlying shell command, not because anything cryptographically prevents a misconfigured or compromised agent from running `git push` anyway. How I Work with Claude Code puts the accountability question directly: "the agent proposes, executes, and explains. You decide, review, and own the result."

That's a structurally different guarantee than RBAC or a LiteLLM virtual key — and it's the exception, not the norm, across AI agent deployments generally. An out-of-scope Kubernetes API

call fails at the API server regardless of what the caller intends. An AI agent asked (or manipulated) to ignore its own written rules is stopped by a permission prompt catching the specific action, or by the human reading the plan before approving it, or it isn't stopped at all — many agent setups skip this entirely in the name of autonomy, which is precisely how the failure modes above happen.

What "good" looks like across all three

The same four questions apply whether the identity behind a request is a person, a workload, or an agent: is the access documented, is it logged, is it scoped to the minimum the identity actually needs, and can it be revoked or rotated without disrupting everything else. Kubernetes RBAC and LiteLLM's virtual keys clear that bar technically — enforcement happens at the API server or gateway, independent of the caller's intent. Today's AI-agent access model, including the one used to write this post, mostly doesn't clear it yet: the controls that exist (permission prompts, written CLAUDE.md rules) are real, but they're closer to policy backed by partial technical friction than to a cryptographic guarantee.

The "documented" bar is easy to treat as the least demanding of the four, but in a regulated environment it can be the one an audit actually turns on — deciding *how* access is granted is a technical decision; keeping a current, explicit record of *that decision* is a separate piece of work, and skipping it doesn't go away just because the technical enforcement is solid. Sometimes that record needs dedicated tooling — an IAM system with built-in policy history, an access-review platform. Just as often, it's enough to document plainly which account — a human user, an AI agent, or a technical/service account — can do what, and why. Either way, that documentation work belongs inside the automated or semi-automated process from the start, not added afterward once someone asks for it: it takes real time, and for the legal and compliance side of an access decision, it isn't optional.

Where this leaves the sovereignty question

This connects directly back to Part 2's six cases. OVH IAM v2, Kubernetes RBAC, and a self-hosted LiteLLM gateway are all things an operator can inspect, configure, and audit themselves — the access model is visible and testable from the outside. Microsoft's "Data Guardian" claim and AWS's "Qualified… Staff" language from Part 2 describe the same kind of access-control intent, but as the *provider's own internal policy* — a customer can read the claim, not inspect or configure the mechanism behind it the way they can with their own RBAC roles. The same technical-enforcement-versus-policy split found inside this blog's own AI-agent tooling shows up again, one layer up, in how hyperscalers describe their own access controls. That distinction — legally- or organizationally-enforced versus technically-enforced — is exactly what the next part of this series takes on directly.

None of this is a recommendation for or against any specific access model. What counts as "good enough" depends on the workload behind it — a low-stakes internal tool and a regulated production system don't need the same bar, and every model covered here (RBAC, IAM, virtual keys, or an AI agent's permission prompts) has a legitimate place depending on what's actually being protected.

Next in this series: Part 4 — Legally vs. Technically Enforced →

Related

- K8s & OpenShift Best Practices: Security — the full RBAC, Pod Security Admission, and service-account reference behind the human- and service-account-access sections
- LLM Inference on OVH MKS: LiteLLM Gateway — per-user virtual keys, budget limits, and audit logging in practice
- Log Archiving Security & Compliance — the OVH IAM v2 section and its IRSA/Workload-Identity gap
- How I Work with Claude Code — the plan-mode and accountability model behind the AI-agent-access section
- Who Builds the Platform? Ownership vs. Stack — Part 2, whose Microsoft/AWS access claims this post's closing section refers back to

Sources

- STACKIT: understand service accounts and service account authentication flows, incl. Workload Identity Federation