09 NOVEMBER 5, 2026 3 PP

Federated identity for agents: how Hypergoose decides who a call is for

Felipe Arce · Tim Child · Collin Rijock

Hypergoose gives every agent and internal app at Exowatt one key and per-identity grants to 20 outside services. Upstream secrets never leave the gateway, and every call carries the person it acts for. The identity model, the bug that shaped it, and what it handles today.

IDENTITY AGENTS GATEWAY OAUTH MCP INFRASTRUCTURE

On September 15 a FluxBot told its owner it was signed in to Zip as a different employee. It was not signed in as anyone. Our Zip connection uses one organization-level API key. The bot's `whoami` tool proved the key worked by fetching `GET /users?page_size=1`. The model saw `whoami` return a person and decided it was that person.

That call went through Hypergoose, the gateway we built so agents and internal apps at Exowatt can reach outside services without holding their secrets. The fix took an afternoon and set the rule we have designed around since: a credential decides what a call may touch, and the gateway decides who the call is for.

Felipe Arce started Hypergoose on September 11, and FluxBots were calling through it that evening. Today it fronts 20 connectors (Slack, Google Workspace, GitHub, Zip, Ramp, our project tracker Tuesday and several internal apps) for 158 agents, 138 of them FluxBots, owned by 80 people. In the last 24 hours it handled 5,195 calls. The gateway stores each upstream secret once, encrypted, and no agent process holds one. Of 360 access grants, 228 activated with no human involved, in a median of 8 ms. The other 132 waited a median of 1.4 minutes for an approver, usually one click on a Slack card.

One key, one host

An agent gets one `hg_` key, and Hypergoose stores only its hash. It calls `/api/gw/:connector/:path` for REST or `/mcp/:connector` for MCP, on the same host. Each call passes these checks in order: the key and agent are active, the connector is enabled, the agent holds a live grant on it, the scope allows the operation, the rate limit has room. A `read` grant allows GET and HEAD on REST; on MCP, Hypergoose hides write tools from `tools/list` and refuses them in `tools/call`.

The gateway picks the credential last, once the call is allowed. It strips any inbound `X-Hypergoose-*` header, swaps the agent's key for the upstream credential, stamps the agent and its owner, and writes one row to an append-only access log in which each row hashes the one before it.

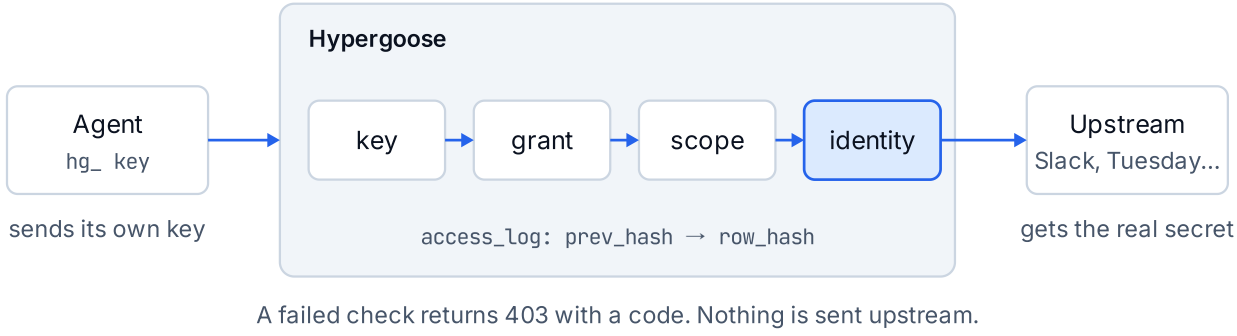


Figure 1. Every call takes the same path. The credential is chosen last, after the access decision, and every call writes one hash-chained log row.

On September 11 we sketched a managed design: Okta-issued tokens, Cedar policies, a credential vault and a broker for Okta's Cross App Access token exchange. The services we needed first do not implement that exchange. We chose a small gateway with keys, two scopes (`read` and `read_write`) and per-connector approvers, with no fine-grained roles in v1. Each finer rule came later, when a real call needed it.

Who a call is for

The Zip fix renamed the tool to `check_connection` and a description telling the model the key represents no person. We made that a rule for all connectors: a tool

may be called `whoami` only if the upstream has a real self endpoint for that credential. FluxBots also got a fenced, platform-filled prompt section naming the human the bot acts for.

Hypergoose has four identity modes.

Mode	Sent upstream	The call acts as
Shared connector (Zip, Ramp)	The connector's one key	A service account
Per-person OAuth (Slack, Gmail)	That person's own token	The person who connected it
Owner identity (Tuesday)	Service key plus a signed token	The bot's verified owner
Delegation (app backends)	Service key plus a signed token	The person the app signed in

Per-person connectors have no shared credential. Setting one returns `409 PER_AGENT_ONLY`, and an agent whose owner has not connected gets `403 IDENTITY_REQUIRED` with the URL to connect. With nothing to fall back to, one person's Slack cannot answer another person's agent. An agent that brings its own credential gets its grant with no approver, because approvals exist to protect shared secrets.

For our own apps, a teammate added signed identity tokens in September: RS256, a 60-second lifetime, a single-use `jti`, and public keys on a JWKS endpoint. Our other two modes reuse that format.

Bots act as their owners

In October a user's bot held an approved `read_write` grant on Tuesday and still got `NOT_FOUND` on her own private space, because Tuesday saw Hypergoose's service user, never invited there. That day we wrote down the policy that fixed it: a bot inherits its owner's access, connections a person makes for themselves need no approval, and approvals are only for shared resources like Ramp and Zip.

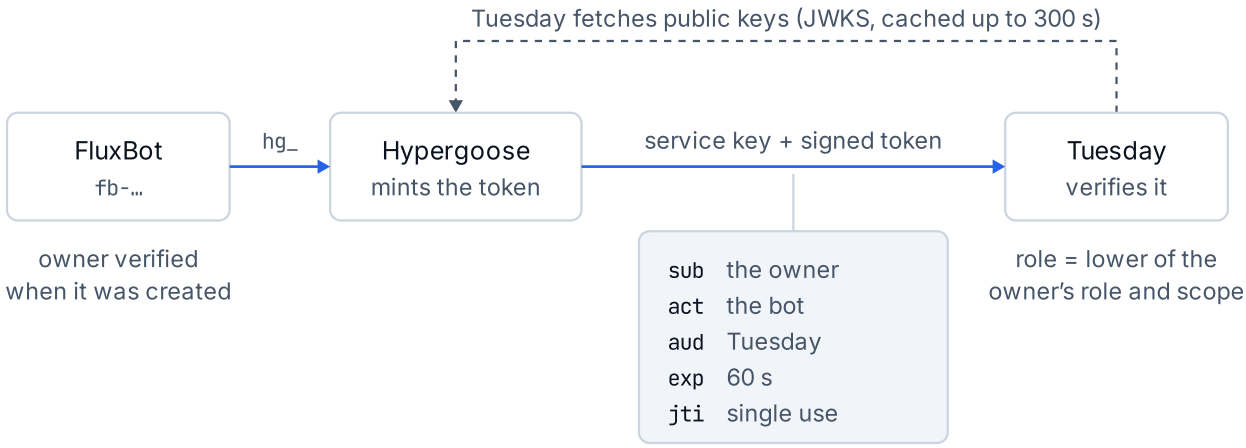


Figure 2. Owner identity. The subject comes from the bot's record on the server; the caller cannot choose it. Tuesday caps the bot at its owner's live role.

Owner identity is a per-connector flag. If a trusted registrar created the bot and it has a verified owner email, Hypergoose mints its token with `sub` set to that owner. The caller cannot set the subject, and Hypergoose refuses any header that names someone else. Delegation uses the same token for an app backend that signed a person in itself; a global admin enables it per connector, and it never falls back to the app's own identity.

A Rust port with proofs

On October 25 we had agents port the Python service to one Rust binary, with the access decisions in a core crate proved with Verus, 5 hours 37 minutes from start to production. A parity rig replayed 4,522 recorded requests against both versions until they all matched, and the core verifies 790 items with no errors. Delete the revoked-grant check from `decide_access` and verification fails. The access log's hash chain verified across 68,022 rows written before the cutover.

The port exposed a leak: the Python HTTP client kept a shared cookie jar, so an upstream's `Set-Cookie` went out again on later requests to that host, for every agent. The Rust client has no jar. The same day we lost 20 minutes of Google sign-in when an automatic dependency bump shipped a JWT library that panics without a crypto backend. A build-only pipeline cannot catch that, so the deploy check needs to sign and verify a real token.

Headroom and next steps

In eight weeks Hypergoose went from an empty repository to about 5,000 calls a day for 158 agents, with no upstream secret in any agent and most grants decided in milliseconds. Turning on `TCP_NODELAY` cut single-client p99 latency from 42 ms to 1.5 ms. The ceiling is now about 800 requests a second, and we think the limit is the advisory lock that serializes the hash chain. That is far above today’s load; if it gets close, I would try one chain per connector first.

Shared bots are the other open piece. Since October 29 a teammate added to a bot gets an agent of their own with grants mirrored and capped at the bot’s level. No team uses it yet. Next we want to move the first shared bot onto it and see where mirrored grants surprise people.