

A Gateway Architecture for Enterprise MCP Authentication:

Unifying Heterogeneous Auth, Identity Delegation, and the User / Non-User Persona Problem

Suraj Kumar* Amy Wang† Srinivasan Manoharan‡

Enterprise AI Platform Engineering

August 2026

Abstract

The Model Context Protocol (MCP) has become the de-facto interface for connecting LLM agents to enterprise tools, and adoption has been explosive: within a year, large organizations went from zero to dozens of internally built MCP servers. That speed created a governance crisis. Each team implemented authentication independently—some with no auth, some with API keys, some with full OAuth—producing a fragmented landscape with no consistent way to authorize callers, track who did what, or offboard a departing employee across the fleet. This paper reports an industry deployment that resolves the crisis with a centralized *MCP gateway*: a single aggregation, governance, and authentication layer that fronts every downstream MCP server.

We make four contributions grounded in production experience. First, we present a **two-axis authentication model** that every MCP server must satisfy—a *persona axis* (interactive *user* vs. automated *non-user*) crossed with a *credential axis* (no-auth, static/dynamic API key, authorization-code-with-PKCE over static/dynamic clients, client credentials, and platform app-context)—all served through a single MCP endpoint. Second, we describe the **gateway authentication layer** itself, which supports three enterprise SSO grants (Authorization Code + PKCE, Device Code, and Resource Owner Password Credentials) and offers callers a choice of token-provisioning models: Bring-Your-Own-Token (BYOT), Generate-Your-Own-Token (GYOT), and full delegated OAuth via RFC 8693 token exchange. Third, we generalize **three end-to-end identity flows**—User→OAuth2, Non-user→Service-Account, and User→Service-Account—that compose client, gateway, and server. Fourth, we document the **deployment evolution** from a CDN/WAF/edge perimeter to private MCP tunnels, and the design of enterprise-wide connectors that let an entire workforce share governed access to AI tools. The architecture is in production, fronting dozens of MCP servers across web, desktop, custom-SDK, and low-code clients.

Keywords: Model Context Protocol, MCP gateway,

enterprise authentication, OAuth 2.0, RFC 8693 token exchange, PKCE, device code, ROPC, service accounts, AI agent identity, zero-trust, API gateway

1 Introduction

1.1 The Governance Crisis Behind MCP Adoption

When the Model Context Protocol [1] was published in late 2024, it solved a real problem elegantly: it gave LLM agents a uniform way to discover and invoke external tools. Adoption inside large enterprises was immediate and uncoordinated. A team that wanted to expose its service to an internal AI assistant could stand up an MCP server in an afternoon. Within months, a single organization could find itself running dozens of internally built MCP servers—for data warehouses, observability platforms, productivity suites, ticketing systems, and low-code automation engines.

The speed of adoption is exactly what created the problem. The MCP specification says implementations *should* support OAuth 2.0 but leaves every detail to the implementer. So every team made its own choice. Some observability servers shipped with no authentication at all, reasoning that they were “internal only.” Others used a single static API key shared across all callers. A few implemented full OAuth flows. The result was a fragmented authentication landscape with three concrete failure modes:

- **No consistent authorization.** Each server decided independently who could call it and with what privileges. There was no central policy.
- **No unified audit.** Each server logged differently, if at all. Answering “which user invoked which tool last Tuesday, and on whose behalf” required correlating incompatible logs across a dozen systems.
- **No clean offboarding.** When an employee left, revoking their access meant contacting every server team individually. There was no single switch.

1.2 A Motivating Example: the Office 365 MCP Server

Consider a concrete server that any enterprise will recognize: an Office 365 MCP server exposing mail, calen-

*suraj.kumar.bh.in@gmail.com

†yuwewang@paypal.com

‡srinivamanoharan@paypal.com

dar, and document tools. It must serve two very different kinds of caller through *the same endpoint*:

1. An **interactive employee** asking an AI assistant to “summarize my unread mail.” This call must run *as that user*: it must see only their mailbox, and the downstream Microsoft Graph token must carry their identity. This requires an interactive OAuth authorization-code flow with the downstream identity provider.
2. A **nightly automation** that compiles a team calendar digest. No human is in the loop. It authenticates with its own service-account credentials (client credentials), and must *not* be able to impersonate any individual user.

A naive design would stand up two servers—one per persona—doubling the operational surface and splitting the tool catalog. Worse, if the automation path is not explicitly constrained, a misconfiguration lets a headless agent obtain a user-scoped token, a textbook confused-deputy escalation [9]. The Office 365 case is not special; every useful enterprise MCP server faces the same two-persona requirement. This single example motivates the entire architecture: one endpoint, two personas, many underlying credential mechanisms, governed centrally.

1.3 Contributions

This is an experience paper. The architecture it describes is deployed in production, fronting dozens of MCP servers used by an entire workforce through shared connectors. Our contributions are:

- **C1 — A two-axis authentication model for MCP servers** (Section 3). We show that the authentication needs of every enterprise MCP server factor cleanly into a *persona* axis (user vs. non-user) crossed with a *credential* axis (no-auth, static/dynamic API key, authorization-code-with-PKCE over static/dynamic clients, client credentials, platform app-context), all served through one endpoint.
 - **C2 — A gateway authentication layer with three SSO grants and three token-provisioning models** (Section 4). The gateway authenticates callers via Authorization Code + PKCE, Device Code, or ROPC, and lets integrators choose Bring-Your-Own-Token (BYOT), Generate-Your-Own-Token (GYOT), or full delegated OAuth via RFC 8693 [2]. The gateway authenticates; downstream servers verify the issued tokens using a shared SDK.
 - **C3 — Three generalized end-to-end identity flows** (Section 5): User→OAuth2, Non-user→Service-Account, and the subtle User→Service-Account case, each composing client, gateway, and server with explicit token handling.
 - **C4 — A deployment and connector evolution report** (Section 7): from a CDN/WAF/edge perimeter with per-vendor IP allowlisting to private MCP tunnels, plus the design of enterprise-wide web and desktop connectors shared across the workforce.
- Throughout, we use diagrams as the primary explana-

tory device: the model is fundamentally about *which token flows where*, and that is far clearer drawn than described. Figure 1 previews the whole system.

2 Background and Problem Framing

2.1 MCP in Brief

MCP defines a client-server protocol in which a server exposes *tools* (callable functions), *resources* (readable data), and *prompts* (templates). The transport is JSON-RPC 2.0 carried over Streamable HTTP (the current recommended transport), HTTP SSE (the original streaming transport), or stdio for local processes. Crucially, the protocol specifies *what* a tool call looks like but says nothing binding about *who* may make it—authentication and authorization are left entirely to the implementer. That single omission is the root of the governance crisis in Section 1.1.

2.2 Why MCP Auth Is Hard: Heterogeneity

The difficulty is not that authentication is unsolved—it is that enterprise MCP servers need *many different kinds* of authentication simultaneously, and the right kind depends on both the downstream system and the calling persona. A data-warehouse server fronts a system that speaks OAuth; an automation engine fronts one that only accepts a static API key; an observability server may need no auth at all. The same server must serve an interactive user (whose downstream calls should run as them) and a headless automation (which must run as a service account). No single credential type covers this space. The contribution of Section 3 is to show that the space is not chaotic—it factors into two clean axes.

2.3 Why a Gateway

Faced with heterogeneity, an enterprise has two choices: make every server team solve auth themselves (the status quo that caused the crisis), or centralize. We centralize, using a *gateway* [8] that fronts every MCP server. The gateway aggregates many servers behind one endpoint (with tool search to avoid context-window bloat from hundreds of tool schemas), enforces guardrails, emits unified observability and metrics, caches and refreshes tokens, and—the focus of this paper—provides the authentication layer that individual servers no longer have to build. Servers are reduced to *verifying* tokens the gateway issues, using a shared SDK. We compare this against the alternatives in Table 1.

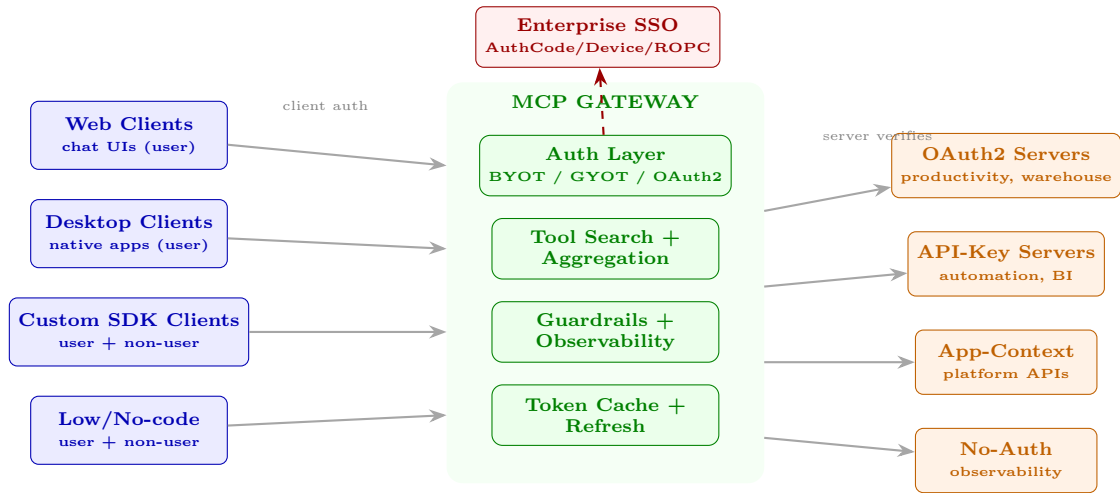


Figure 1: System overview. Heterogeneous clients authenticate *once* to the gateway, which authenticates them against enterprise SSO and resolves the correct downstream credential per MCP server. The gateway is the single point for auth, aggregation, guardrails, observability, and token caching. Downstream servers only *verify* gateway-issued tokens via a shared SDK.

Table 1: Per-server auth vs. centralized gateway.

Dimension	Per-server	Gateway
SSO registrations	N (one each)	1
Audit trail	Fragmented	Unified
Offboarding	Contact each team	Central revoke
Token caching	Re-implemented	Shared
Tool-context bloat	N catalogs	Tool search
Guardrails	Per-team	Centralized

3 The Two-Axis Authentication Model

Our central design claim is that the seemingly chaotic space of enterprise MCP authentication factors into two orthogonal axes, and that a single MCP endpoint can serve the entire matrix. The **persona axis** distinguishes an interactive *user* from an automated *non-user* (batch jobs, scheduled pipelines, agents with no human in the loop). The **credential axis** enumerates the concrete mechanisms. Figure 2 shows the full model; the persona determines which credential mechanisms are *available*, and the downstream system determines which is *used*.

3.1 User Persona

An interactive user is a human driving an AI client. The server can authenticate them in one of three ways, in increasing order of identity fidelity:

- **No auth:** used only for low-sensitivity observability platforms where the data carries no per-user authorization. The gateway still records the caller; the *server* simply does not require a credential.
- **API key:** either a **static** key (a single long-lived secret, e.g. a low-code automation engine) or a **dynamic** key constructed per-user from stored creden-

tials (e.g. a BI/analytics platform that mints a short-lived key bound to the requesting user).

- **Authorization code with PKCE:** the high-fidelity path—an interactive OAuth flow that produces a downstream token carrying the user’s identity. The downstream OAuth client may be **static** (one shared client registration, e.g. a productivity suite or data warehouse) or **dynamic** (a client registered per tenant or per integration, typical of multi-tenant SaaS such as a ticketing platform or an observability platform that issues each customer its own OAuth client).

3.2 Non-User Persona

A non-user caller is an automation with no human present. Its available mechanisms differ because there is no interactive consent step and no user identity to carry:

- **No auth:** the same low-sensitivity observability case.
- **API key:** **static** (automation, ticketing, or observability platforms that accept a service key) or **dynamic** (a BI/analytics platform that mints a short-lived key for the service account).
- **Client credentials:** the OAuth machine-to-machine grant, used where the downstream system speaks OAuth and can accept a service-account token (e.g. a productivity suite via its directory’s client-credentials grant).
- **Platform app-context:** a platform-native machine identity (e.g. an internal app-context credential for ops/SRE tooling) where the downstream system trusts a derived application context rather than a bearer token.

Key insight (C1). The persona axis is not cosmetic—it is a *security boundary*. Authorization-code-with-PKCE is available to users (it requires interactive consent) but *not* to non-users; client creden-

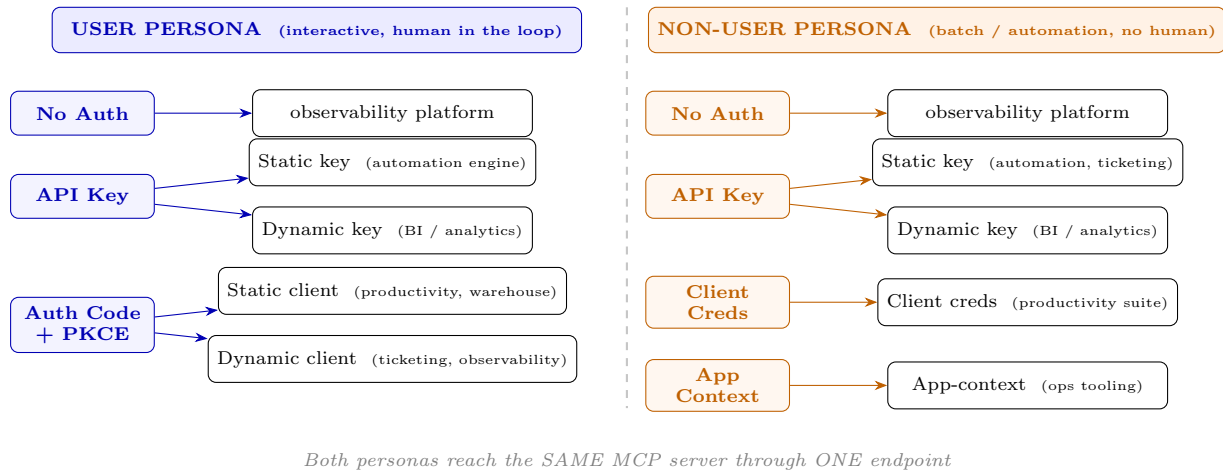


Figure 2: The two-axis authentication model. Each MCP server supports both personas through a single endpoint. The persona (user vs. non-user) gates which credential mechanisms are available; the downstream system determines which is actually used. Examples in parentheses are representative server classes.

tials and app-context are available to non-users but never carry a user identity. This asymmetry is what prevents a headless automation from impersonating a human.

3.3 One Endpoint, Both Personas

The architectural commitment is that a single MCP server endpoint serves both personas. The server does not run two ports or two catalogs. Instead, the *token presented* determines the persona, and the gateway—not the server—resolves the appropriate downstream credential. Table 2 summarizes which mechanisms each persona may use.

Table 2: Credential mechanisms available per persona. ✓ = available; — = not available by design.

Credential mechanism	User	Non-user
No auth	✓	✓
Static API key	✓	✓
Dynamic API key	✓	✓
Auth code + PKCE (static client)	✓	—
Auth code + PKCE (dynamic client)	✓	—
Client credentials	—	✓
Platform app-context	—	✓

4 The Gateway Authentication Layer

4.1 Design Evolution: How We Got Here

The gateway’s auth design was not chosen up front; it evolved as the deployment grew. Telling that story is the clearest way to motivate the final design.

Stage 0 — the user flow only. The first deployments served interactive users exclusively. People wanted an AI assistant that could read their data

as them, so the obvious first build was an interactive OAuth flow: the user logs in through the enterprise SSO, and the gateway obtains a token on their behalf. This worked, and for a while the only persona was the human user.

Stage 1 — the need for automation. Demand quickly arrived for non-interactive usage: nightly digests, batch enrichment, agentic workflows that run unattended. These have no browser and no human to complete a consent screen. The interactive flow could not serve them. We added a machine path—service-account credentials—and with it the *non-user* persona was born. This is the moment the two-axis model became necessary.

Stage 2 — heterogeneity bites. As more servers came online, we hit the heterogeneity wall of Section 2.2: different downstream systems demanded different credential types. Rather than let each server reinvent auth, we pushed all of it into the gateway and reduced servers to token *verifiers*.

Stage 3 — delegation. Finally, agentic clients made identity delegation concrete: a user asks an agent to act, the agent calls the gateway, and the audit log must show *both* the user and the agent. Standard OAuth carries one subject; we adopted RFC 8693 token exchange to carry both. Figure 3 summarizes the arc.



Figure 3: Design evolution. Each stage added a requirement the previous design could not meet, driving toward the final centralized, delegation-aware gateway.

4.2 Three Enterprise SSO Grants

The gateway authenticates callers against the enterprise SSO using three OAuth grant types, matched to how the caller runs:

- **Authorization Code + PKCE** — for *user* personas with a browser (web and desktop clients). The

standard interactive flow; PKCE [3] protects the authorization code in transit and is mandatory for all clients.

- **Device Code** — for *user* personas on input-constrained or remote machines (e.g. a CLI on a headless dev box). The user authorizes on a second device; the originating machine polls for the token.
- **Resource Owner Password Credentials (ROPC)** — for *non-user* personas: service accounts that present credentials directly. ROPC is restricted to machine identities and never used for interactive humans.

Table 3: SSO grant selection by client and persona.

Caller	Persona	Grant
Web / desktop client	User	Auth Code + PKCE
Remote / headless machine	User	Device Code
Service account	Non-user	ROPC

4.3 Three Token-Provisioning Models

A key flexibility of the gateway is that integrators are not forced into one token model. We support three, covering the full spectrum from “the client already has a token” to “the gateway brokers everything”:

- **BYOT (Bring Your Own Token)**. The client already holds a valid SSO token and presents it. The gateway validates via introspection and uses it. Lowest friction when the client is already SSO-integrated.
- **GYOT (Generate Your Own Token)**. The client asks the gateway to initiate a grant on its behalf (e.g. start a device-code flow), and the gateway returns the resulting token. Useful for clients that can drive a flow but do not want to implement OAuth themselves.
- **Full delegated OAuth (RFC 8693)**. For agent-with-user scenarios, the agent exchanges its user-scoped token for a gateway-scoped token in which the SSO injects the *act* (actor) claim. This is the only model that preserves both identities; we detail it in Section 5.

Key insight (C2). Offering BYOT, GYOT, and full OAuth side by side is what lets a single gateway serve clients ranging from fully SSO-native web apps to thin custom-SDK agents—without forcing any of them to re-implement the others’ machinery.

4.4 Two Ways to Obtain a Static-Client Token

The authorization-code-with-PKCE static-client case (e.g. a productivity suite, data warehouse, or SRE server) deserves special note because the gateway supports *two* distinct ways to obtain the downstream token:

1. **Direct token exchange via the browser path.** The user completes an interactive flow against the downstream IdP directly through the browser.

2. Token exchange without the browser path.

The gateway takes the user’s existing SSO token (already obtained via Authorization Code + PKCE) and performs an RFC 8693 token exchange to mint the downstream token—no second browser round-trip.

The second path is strictly better for user experience when the user is already authenticated to the gateway: it removes a redundant consent screen while preserving full identity fidelity. Figure 4 contrasts the two.

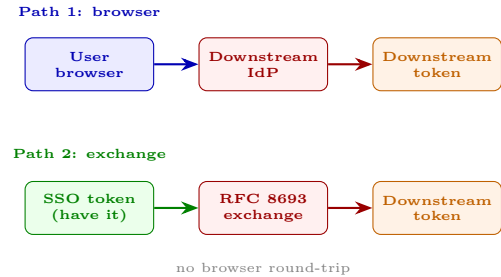


Figure 4: Two ways to obtain a static-client downstream token. Path 2 reuses the user’s existing SSO token via RFC 8693, avoiding a redundant browser consent.

4.5 Gateway Issues, Server Verifies

A clean separation of duties underpins the whole design: the **gateway authenticates** and the **downstream server verifies**. Servers do not implement OAuth; they include a shared SDK that validates gateway-issued tokens (signature, audience, expiry, and introspection). The gateway also handles token **caching** and **refresh** so that downstream systems are not hammered with introspection calls and users are not repeatedly prompted. This is what makes onboarding a new MCP server a matter of registering it and adding the SDK, rather than building an identity stack.

5 Three End-to-End Identity Flows

The two-axis model (Section 3) and the gateway layer (Section 4) compose into three end-to-end flows that span client, gateway, and downstream server. We present them generalized from a concrete data-warehouse server, but they apply to any MCP server. The three flows are distinguished by *whose identity reaches the downstream system*.

5.1 Flow 1 — User → OAuth2

A human user runs as themselves end-to-end. The client completes an interactive SSO authorization-code flow. Crucially, the gateway does *not* hand the raw OAuth2 token back to the client; instead it stores the token internally and returns an opaque *session handle*. On every subsequent MCP call the client presents the session, and the gateway looks up (and silently refreshes, if expired) the token mapped to that session before validating it

and forwarding the call. Keeping the token server-side means it never transits the client and can be refreshed without re-prompting the user.

Listing 1: Flow 1 — the OAuth2 token the gateway holds (mapped to the session). `act` is absent because the user authenticated directly.

```
{
  "sub": "user@corp.com",
  "aud": "https://gateway.example.com",
  "scope": "mcp:tools",
  "act": null           // absent: direct user,
                        // no delegation
}
```

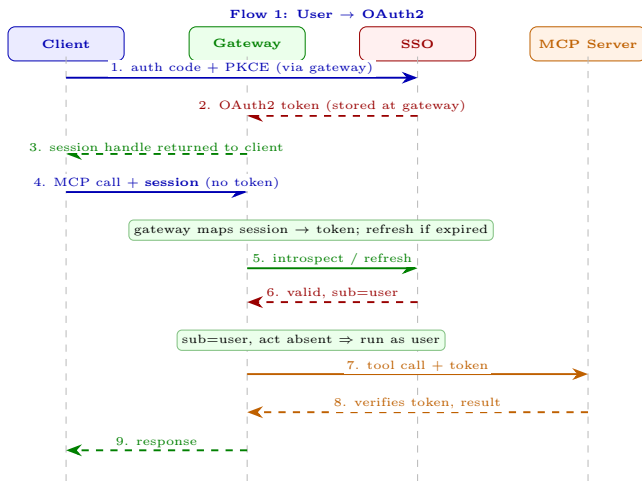


Figure 5: Flow 1. The user authenticates interactively. The gateway keeps the OAuth2 token server-side and returns an opaque *session* to the client (step 3). Subsequent calls carry the session, not the token; the gateway resolves and refreshes the mapped token (step 4–5) before forwarding. The user’s identity is preserved end-to-end.

5.2 Flow 2 — Non-user → Service Account

An automation runs as a service account. The agent authenticates via ROPC and presents *only* its service-account credential on the MCP call. The downstream API key is **not** passed by the client—it is *provisioned to the gateway ahead of time* (out of band, during server onboarding) and the gateway attaches it when forwarding. No user identity is involved at any point, and the downstream secret never transits the client.

5.3 Flow 3 — User → Service Account

The subtle case. A human user is present, but the downstream system is accessed via a service account—*provided the user is authorized to use that service account*. The client completes an interactive SSO flow (so the gateway knows the real user), the gateway *validates that the user has service-account access*, and only then forwards the call under the service-account credential. This composes user accountability with service-account execution, and the authorization check is the crux.

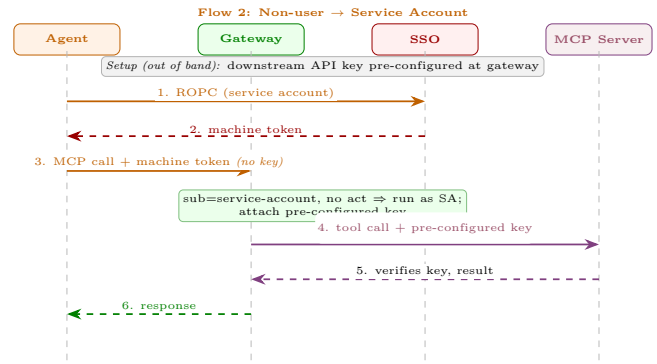


Figure 6: Flow 2. A headless agent authenticates via ROPC and sends only its machine token. The downstream API key is provisioned to the gateway ahead of time (grey note), not passed on the call; the gateway attaches it when forwarding. No user identity is present and the downstream secret never reaches the client.

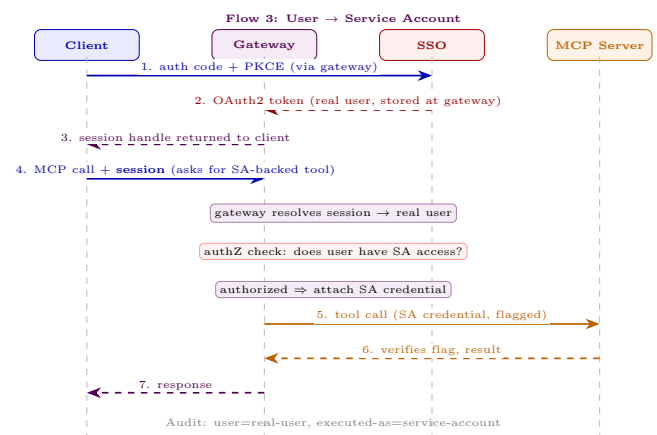


Figure 7: Flow 3. A real user authenticates; the gateway stores the token and returns a session (step 3). The client passes the session (step 4); the gateway resolves the real user, then performs the user-has-SA-access check *before* attaching the service-account credential. The audit record captures both the human and the execution identity.

Key insight (C3). Flow 3 is where most naive designs fail. Letting a user trigger a service-account-backed tool without an explicit authorization check is the confused-deputy escalation in disguise. The gateway’s user-has-SA-access check, performed *before* attaching the service-account credential, is the control that makes this flow safe.

5.4 Delegation: Preserving Both Identities

When the caller is an *agent acting for a user*, Flow 1 is extended with RFC 8693 token exchange. The agent first obtains a user-scoped token (AT1), then exchanges it for a gateway-scoped token (AT2) in which the SSO injects an `act` claim naming the agent. The result carries both identities in one signed token:

Listing 2: Delegated token (AT2) — both identities in one signed JWT. `sub` is the user (preserved from AT1); `act.sub` is the agent (injected by the SSO during exchange).

```
{
```

```

"sub": "user@corp.com",           // user, preserved
"act": {
  "sub": "agent-prod"             // agent, injected
    by SSO
},
"aud": "https://gateway.example.com",
"scope": "mcp:tools"
}

```

The critical configuration detail—and the single most common misconfiguration—is that the subject-token validation must check the audience of AT1 (the agent’s own client ID), *not* the gateway audience. AT1 is scoped to the agent application; the gateway is the *target* of the exchange, not the current audience. Multi-agent chains extend this automatically: each hop appends one level of `act` nesting, producing a complete, cryptographically bound delegation trail in every token that reaches the gateway.

6 Authorization Across Caller Types

Authentication establishes *who* is calling; authorization decides *whether they may*. A recurring source of confusion in MCP deployments is conflating the two. Our design keeps them distinct and places each authorization decision at a well-defined point. This section makes the authorization model explicit across the cases of Sections 3–5.

6.1 Two Enforcement Points

Authorization is enforced at two layers, and both matter:

- **Gateway authorization** — coarse-grained, identity-level decisions made before the call is forwarded: is the caller permitted to use this server at all; for a user, do they hold the required role; for the User→SA case, are they entitled to the service account.
- **Downstream authorization** — fine-grained, resource-level decisions made by the downstream system using the forwarded credential: which records, which mailbox, which dataset. The gateway never tries to replicate this; it forwards an appropriately scoped credential and lets the system of record decide.

Design rule. The gateway answers “*may this identity reach this tool?*”; the downstream system answers “*may this identity touch this data?*” Splitting the decision this way keeps the gateway free of per-resource policy while still enforcing a hard identity boundary.

6.2 Decision by Caller Type

Table 4 summarizes who decides and what is checked for each caller type. The asymmetry from Section 3 reappears here as an authorization rule, not just an availability rule: a non-user identity can never satisfy a user-scoped authorization check, because it carries no user identity to check.

Table 4: Authorization decision points by caller type.

Caller	Gateway checks	Downstream checks
Direct user	user role / scope for the tool	per-resource, as the user
Agent-with-user	user role (agent inherits, never exceeds)	per-resource, as the user
Autonomous agent	service-account role; user-scoped patterns denied	per-resource, as the service account
User → SA	user-has-SA-access before attaching SA credential	per-resource, as the service account

6.3 The User → Service-Account Check

The most security-sensitive authorization decision is the User→SA case (Flow 3). Here a human triggers a tool that executes under a service account. If the gateway forwarded the SA credential without checking, any user who could reach the tool would inherit the service account’s full privileges—a confused-deputy escalation [9]. The gateway therefore performs an explicit entitlement check (“is this user authorized to act through this service account?”) *after* resolving the user from their session and *before* attaching the credential. The decision and both identities are written to the audit log, so every SA-backed action is traceable to the human who initiated it.

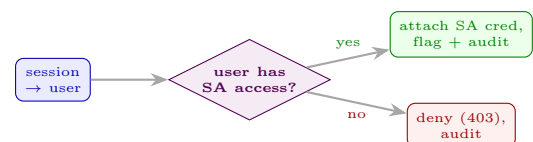


Figure 8: The User→SA authorization gate. The entitlement check runs after the session resolves to a real user and before any service-account credential is attached.

7 Clients, Connectors, and Deployment

7.1 Four Categories of Client

The gateway is consumed by four client categories, each spanning different personas:

- **Web clients** (browser-based AI chat UIs) — user persona, via Authorization Code + PKCE.
- **Desktop clients** (native AI apps) — user persona, also via Authorization Code + PKCE.
- **Custom-SDK clients** (built on agent frameworks such as CrewAI, the Claude Code SDK, the OpenAI Agents SDK, or LangChain, using Streamable-HTTP MCP transport) — *both* user and non-user

personas, since these frameworks drive interactive sessions and unattended batch jobs alike.

- **Low-code / no-code clients** (workflow builders) — *both* personas, for the same reason.

Table 5: Client categories and the personas they exercise.

Client category	User	Non-user
Web (chat UIs)	✓	—
Desktop (native apps)	✓	—
Custom SDK (agent fwks)	✓	✓
Low/no-code (workflows)	✓	✓

7.2 Enterprise-Wide Connectors

A practical contribution of the deployment is *shared enterprise connectors*: a single web connector and a single desktop connector for the gateway, configured once and distributed to the entire workforce. An employee installs the connector and immediately has governed access to every MCP server the gateway fronts, authenticated through corporate SSO, with no per-server setup. This is what turns a back-end architecture into something an entire organization actually uses—and it is only possible because authentication is centralized at the gateway.

7.3 Perimeter Evolution: From Edge to Tunnel

Exposing the gateway to web clients raised a network-security question with two answers, and we evolved from one to the other.

Starting point — CDN / WAF / edge route. We initially fronted the gateway with a CDN, a Web Application Firewall (with rules to block SQL-injection-style payloads), and an edge route with a per-vendor IP allowlist. This works but is operationally heavy: every new client vendor requires an allowlist change, and the WAF must be tuned to avoid blocking legitimate MCP traffic.

Direction — private MCP tunnels. We are migrating to private MCP tunnels [14], in which the client establishes an outbound tunnel to the gateway rather than the gateway exposing a public ingress. This removes the need for per-vendor IP allowlisting and shrinks the public attack surface. Critically, the tunnel does not change the auth model: every request inside the tunnel still carries a fresh token that the gateway validates independently. The tunnel secures *transport*; the token secures *authorization*. Figure 9 contrasts the two.

8 Discussion and Lessons Learned

The hard part was organizational, not cryptographic. The OAuth standards involved—PKCE, device code, ROPC, RFC 8693—are mature and well-

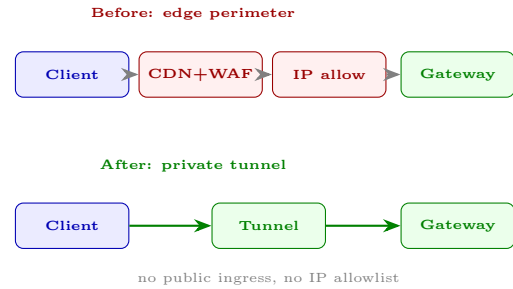


Figure 9: Perimeter evolution. The tunnel removes per-vendor IP allowlisting and public ingress while leaving per-request token validation unchanged.

implemented by enterprise SSO products. The genuinely hard work was getting every server team to route through the gateway instead of rolling their own auth, enabling token exchange on the SSO (a feature the identity team had never previously needed), and agreeing on the non-user-cannot-impersonate-user boundary.

The two-axis model is the reusable idea. Teams initially saw a bewildering list of auth mechanisms. Reframing it as persona $\times$ credential turned an unbounded problem into a small matrix that a new server team can place themselves into in minutes.

Centralization pays for itself at offboarding and audit. The first time a security review asked “what did every agent do on behalf of every user last month,” the unified gateway log answered in one query—something that was impossible in the per-server world.

Tool search matters as much as auth. Aggregating dozens of servers behind one endpoint would flood an agent’s context with hundreds of tool schemas. Gateway-side tool search keeps the agent’s working set small, which is a precondition for the single-endpoint model to be usable at all.

9 Related Work

MCP security. Recent work surveys enterprise MCP risks and mitigation frameworks [10], catalogues content-injection and overstepping-agent adversaries [11], and measures ecosystem registry-vetting weaknesses [12]. A concurrent line addresses server-side zero-trust authorization and permission-filtered tool discovery [13]. These works identify the threat space; our contribution is a deployed gateway architecture that unifies the heterogeneous *authentication* layer those works assume exists.

Zero-trust and identity. NIST SP 800-207 [6] formalizes zero-trust; BeyondCorp [7] demonstrates enterprise-scale deployment. We apply these principles at the MCP boundary, validating every request regardless of tunnel establishment.

OAuth standards. Our design composes token introspection [4], token exchange [2], PKCE [3], and resource indicators [5] into a single coherent gateway flow. Agent-identity delegation via the *act* claim follows emerging identity-for-AI guidance.

API gateways. Service meshes and API gateways [8] terminate auth at the HTTP boundary but do not address MCP-specific concerns: the two-persona-single-endpoint requirement, downstream credential heterogeneity, or tool-search aggregation.

10 Conclusion

Enterprise MCP adoption outran governance: dozens of independently built servers, each with its own auth, produced a landscape no one could authorize, audit, or off-board cleanly. We described a production gateway that resolves this with a single authentication and aggregation layer. The reusable ideas are a two-axis authentication model (persona $\times$ credential, one endpoint), a gateway layer offering three SSO grants and three token-provisioning models (BYOT, GYOT, full OAuth via RFC 8693), and three generalized identity flows—including the easily-mishandled User $\rightarrow$ Service-Account case. The architecture is in production across web, desktop, custom-SDK, and low-code clients, fronting dozens of MCP servers through enterprise-wide shared connectors, and is migrating from an edge perimeter to private tunnels without changing its per-request token model. We believe the two-axis model and the gateway-issues/server-verifies split are directly transferable to any organization facing the same MCP governance crisis.

References

- [1] Anthropic. 2024. *Model Context Protocol Specification*. <https://modelcontextprotocol.io>
- [2] M. Jones, A. Nadalin, B. Campbell, J. Bradley, C. Mortimore. 2020. *OAuth 2.0 Token Exchange*. RFC 8693, IETF.
- [3] N. Sakimura, J. Bradley, N. Agarwal. 2015. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636, IETF.
- [4] J. Richer. 2015. *OAuth 2.0 Token Introspection*. RFC 7662, IETF.
- [5] B. Campbell, J. Bradley, H. Tschofenig. 2020. *Resource Indicators for OAuth 2.0*. RFC 8707, IETF.
- [6] S. Rose, O. Borchert, S. Mitchell, S. Connelly. 2020. *Zero Trust Architecture*. NIST SP 800-207.
- [7] R. Ward, B. Beyer. 2014. BeyondCorp: A New Approach to Enterprise Security. *login:* 39, 6.
- [8] W. Li, Y. Lemieux, J. Gao, Z. Zhao, Y. Han. 2019. Service Mesh: Challenges, State of the Art, and Future Research Opportunities. *IEEE SOSE*.
- [9] N. Hardy. 1988. The Confused Deputy. *ACM SIGOPS Operating Systems Review* 22, 4.
- [10] V. S. Narajala, I. Habler. 2025. Enterprise-Grade Security for the Model Context Protocol. *arXiv:2504.08623*.
- [11] H. Errico, J. Ngiam, S. Sojan. 2025. Securing the Model Context Protocol: Risks, Controls, and Governance. *arXiv:2511.20920*.
- [12] X. Li, X. Gao. 2026. A First Look at the Security Issues in the MCP Ecosystem. *IEEE/IFIP DSN*.
- [13] Anonymous. 2026. Zero-Trust Authorization and Discovery for Enterprise MCP. *ACM AISec*.
- [14] Anthropic. 2026. *MCP Tunnels: Overview*. <https://platform.claude.com/docs/en/agents-and-tools/mcp-tunnels/overview>