

Zero-Trust Authorization and Discovery for Enterprise MCP

Huan Li
PayPal Inc.
USA

huanli1@paypal.com

Yuwei Wang
PayPal Inc.
USA

yuwewang@paypal.com

Srinivasan Manoharan
PayPal Inc.
USA

srinivmanoharan@paypal.com

Abstract

LLM agents translate natural-language context, which may include attacker-controlled text, into privileged tool calls, so authorization must remain effective even when an agent is prompt-injected or otherwise adversarially steered. The Model Context Protocol (MCP) has become a widely adopted interface for this boundary, yet the authentication and authorization primitives in its official SDKs fall short of enterprise zero-trust requirements—most acutely the *dual-persona* model, in which one server must serve human users (corporate SSO) and automated agents (service-account credentials in a different header) at once. We conduct a systematic gap analysis of six surveyed MCP SDKs (Python, TypeScript, Go, Rust, C#, Swift) and identify three structural shortcomings: Authorization-header-bound credential extraction that makes dual-persona deployment difficult without replacing SDK-level middleware, the absence of pre-authentication tool discovery, and the lack of fine-grained per-tool authorization in the base SDKs. We close these gaps with composable extensions to FastMCP—cross-header credential normalization for enterprise deployments serving both human and service-account callers, cached token verification across heterogeneous IdPs, an unauthenticated metadata endpoint for credential-free registry discovery, and permission-filtered tool visibility kept consistent with per-tool invocation enforcement by a single declarative annotation—all implemented without modifying the protocol or SDK internals. Across four frontier LLMs over 2160 attempts, an in-body-check-only server still exposes forbidden tools (152/720, 21.1%; per-model rates 0% to 50% on identical prompts), whereas permission-aware visibility drives the rate to 0/720; visibility-only filtering remained bypassable by scripted clients, while models continued to reference the hidden tool by name in up to 94% of settings where the injected prompt let the model infer that name, confirming that discovery controls cannot replace invocation-time enforcement.

Keywords

Model Context Protocol, zero-trust architecture, AI agent security, authentication, authorization, MCP SDK

1 Introduction

LLM agents introduce an authorization adversary that traditional API security was not designed to contain. Their privileged tool calls are generated from natural-language context that may include untrusted text: retrieved documents, third-party tool outputs, user prompts, or attacker-controlled web content. The agent thus becomes a vehicle for indirect prompt injection [12, 40]: network-layer authentication can verify the credential on a request, but not whether malicious context shaped the requested action. The attacker is not at the transport boundary; the attacker is *inside the caller’s reasoning loop*. Authorization must therefore be enforced

at the MCP tool-invocation layer, before model-generated requests reach concrete privileged operations.

The Model Context Protocol (MCP) [3], introduced in late 2024, has become a widely adopted interface where this enforcement boundary lives. MCP standardizes how LLM agents discover and invoke external tools, prompts, and resources. Within just over a year of release, enterprises are deploying dozens to hundreds of MCP servers exposing privileged operations: querying production databases, triggering CI/CD pipelines, managing Kubernetes clusters, and reading proprietary codebases. Each registered tool is a new attack surface: an agent with deployment access can deploy code; one with database access can exfiltrate data. Unlike a human API caller, the agent may attempt these operations because an attacker-controlled string in its context window told it to. This risk is catalogued in the 2025 OWASP LLM Top-10 as “Excessive Agency” [29] and flagged as an emerging surface in the NIST AI Risk Management Framework [27].

This work positions zero-trust enforcement at the MCP boundary as a guardrail for agents. A correctly configured MCP server should reach the same per-tool authorization decision whether the calling agent is benign, hallucinating, jailbroken, or actively prompt-injected: the agent’s instructions cannot grant it scope it does not already hold. This complements prompt-injection defenses that operate *inside* the agent [33, 40]. Those defenses make the agent more likely to refuse a malicious instruction; ours makes that refusal unnecessary, because even if the agent complies, the server-side authorization layer rejects the call.

The zero-trust model, formalized in NIST SP 800-207 [32], provides the right framework for this. Zero-trust rejects implicit trust regardless of network location and is widely deployed to secure human users [25, 37]. Applied to LLM-agent tool protocols, four requirements follow:

- (1) Every tool call, not just the initial connection, must carry verified credentials.
- (2) Each agent must see and invoke only the components it is authorized to use, so prompt-injection cannot enlarge the tool-calling vocabulary.
- (3) Control planes (registries, catalogs) must be able to discover a server’s tools and auth methods *without* authenticating, so they can index many servers credential-free.
- (4) Multiple identity providers must coexist, because enterprises run heterogeneous identity stacks across human users and service-account automation.

Gap. A systematic review of the six MCP SDKs we surveyed (Section 2.3) found none that satisfies these requirements through documented extension paths at the versions we analyzed (Section A). The three earlier SDKs (Python, TypeScript, Go) default to Authorization-header credential extraction, so serving both human and automation credentials on one server requires replacing

SDK-level middleware rather than configuring it. The three newer SDKs (Rust, C#, Swift) now expose authentication extension points, but in our versioned snapshot none provides both registry-oriented pre-auth tool metadata and SDK-native declarative per-tool authorization. FastMCP now offers scope-based component authorization governing both visibility and direct invocation (Section 6), so we do not claim novelty for those two surfaces. Existing work has examined MCP security through gateway-level centralized authentication and governance [19, 24], and the MCP specification’s own Enterprise-Managed Authorization extension [4] now standardizes IdP-mediated connection admission, leaving resource-level authorization to the server; other work covers ecosystem-scale threat measurement [11, 21, 26], and host-side runtime defenses [39]. Rather than claiming the first MCP authorization mechanism, we focus on a three-surface consistency problem: deriving sanitized pre-auth metadata, authenticated listings, and invocation checks from one server-local policy declaration.

Why MCP-agent authorization differs. Authentication, RBAC, and token introspection are mature areas with well-known patterns. The contribution we claim is not the individual primitives but their composition to address three properties of enterprise MCP-agent deployment:

(a) *The caller is conditioned on attacker-controlled text.* A traditional API client’s intent is fixed by its code path. An LLM-agent client’s next request is generated from a context that may contain retrieved documents, third-party tool outputs, or attacker-supplied web content. Per-request intent is therefore adversarially mutable [12, 22, 30], so authorization must hold uniformly over all natural-language inputs the agent could receive, not over a presumed-benign caller. The AgentDojo benchmark [9] reports significant attack success on aligned models even under sandbox constraints, evidence that input-side defenses alone do not converge.

(b) *Discovery is part of the attack surface.* Opaque REST APIs ship documentation out-of-band; clients invoke documented endpoints. MCP makes discovery a runtime channel: the agent’s tool-calling vocabulary is the set of tools `list_tools` returns. A listing not filtered by permission therefore exposes tools the caller cannot use and widens what the agent will attempt. The listing is thus part of the attack surface: visibility should track invocability—an agent should not be listed a tool it cannot invoke.

(c) *One server serves both human and automation principals.* A single tool catalog often fronts both human SSO and service-account automation. This requirement is not unique to MCP—conventional API stacks handle it at the gateway [20]—but the MCP SDKs bind credential extraction to the `Authorization` header and offer no documented way to compose backends across headers, so a dual-persona server requires replacing SDK middleware (Section 2.3). Standing up parallel servers per credential type instead bifurcates the catalog and doubles operational surface.

These properties shape where enforcement can live. Fine-grained, per-tool authorization is an application-layer concern in HTTP services too—a gateway handles authentication and coarse, path-level policy, while resource-level decisions sit in the application—so this part is not MCP-specific. What is particular to MCP is that the available tools are themselves a runtime protocol response: filtering `list_tools` by caller permission requires parsing MCP’s

JSON-RPC semantics, which is naturally done in the MCP layer rather than at a generic gateway. Keeping that per-role discovery consistent with per-tool enforcement is the work our contributions address (Section 6).

Contributions. We realize four such extensions, labeled **C1–C4** below, unified by one thesis: *tool visibility is part of the agent’s security boundary*. The tools surfaced through `list_tools` and the pre-auth metadata endpoint shape the agent’s reachable action space, so authorization must hold over discovery and invocation alike, from a single declaration. Each contribution maps to one or more NIST SP 800-207 zero-trust tenets and defends a class of adversary goal in our threat model (Section 2.4):

- **C1. Cross-header credential normalization.** We independently validate human SSO and service-account credentials arriving on distinct headers and normalize whichever succeed into one canonical principal. With an upstream gateway [19], this defines the server-side trust contract without relying on network position (Section 6).
- **C2. Cached verification across heterogeneous IdPs.** Many enterprise IdPs issue *opaque* tokens that must be validated by an API round-trip rather than locally (as with JWT/JWKS). We cache verified results within a bounded TTL, so “always verify” holds without a round-trip per call; the resulting staleness-versus-revocation tradeoff is analyzed against G4.
- **C3. Control-plane/runtime-plane separation via pre-auth discovery.** An unauthenticated metadata endpoint lets a registry index many servers’ tools and auth methods without holding credentials, so compromising it grants no power to call tools.
- **C4. Cross-surface policy consistency.** One per-tool declaration governs both visibility and invocation. Empirically, we show why both are necessary: filtering alone remains directly bypassable and may still lead LLMs to name hidden tools (Section 4.3).

The extensions plug into native framework extension points at three layers: HTTP middleware, FastMCP application middleware, and function decorators (Figure 1).

We evaluate the design from two angles. Adversarially, permission-aware visibility holds attempt rates at zero across four frontier LLMs whose baseline behavior varies sharply by vendor (Section 4.3), showing the stable boundary is architectural rather than a property of any one model’s alignment. Operationally, the extensions run in production on an enterprise AI-agent platform adopted by dozens of authenticated MCP servers (Section 4.2), the setting where the dual-persona and registry-indexing patterns motivating C1 and C3 arise directly.

2 Background and Motivation

2.1 The Model Context Protocol

MCP defines a client-server architecture in which servers register three component types [3]:

- **Tools:** executable functions an agent can invoke, such as querying a database or triggering a deployment.
- **Prompts:** reusable prompt templates with arguments.

- **Resources:** read-only data sources (files, schemas, documents).

Transports include Streamable HTTP and stdio for local processes (an earlier SSE-based transport predates Streamable HTTP but has been deprecated since the 2026-07-28 revision [5]). HTTP-based transports can leverage standard web security primitives; stdio cannot.

The specification says implementations “SHOULD support OAuth 2.0” but leaves the specifics to each SDK. It defines OAuth discovery endpoints (RFC 8414 Authorization Server Metadata [18] and RFC 9728 Protected Resource Metadata [16]) but provides no mechanism for tool-level authorization, pre-auth schema discovery, or multi-provider authentication.

2.2 Motivating Scenarios

Dual-persona deployment. A single enterprise MCP server exposes one tool catalog to two populations: humans authenticating with corporate SSO (a Bearer token in `Authorization`) and automation authenticating with service-account credentials in a different header. Standing up two parallel servers doubles the operational surface and bifurcates the catalog. This is the anchor scenario for our work and is not available through any documented extension path in the six MCP SDKs we surveyed.

Registry-driven catalog of MCP servers. As MCP adoption scales, enterprises stand up an MCP registry: a control-plane platform that indexes many servers and, for each, publishes a unified catalog of their tools, prompts, and resources (with schemas and permission requirements) and the auth methods accepted. The registry is not a runtime user, so reusing the agent’s authenticated `list_tools` path fits it poorly; a pre-auth metadata endpoint serves it directly (Section 4.3.5).

Role-based tool visibility. An SRE server may expose both diagnostic and remediation tools; on-call engineers should see everything, juniors only diagnostics. Base SDK handlers commonly return the full registered list regardless of caller, so this filtering must happen at the application middleware layer (C4)—which, beyond access control, also keeps the agent from attempting tools it cannot use and trims the schemas carried in its context (Section 4.2). This visibility boundary—keeping a tool out of the schema the agent is shown—is the property our adversarial evaluation centers on (Section 4.3.1).

2.3 Cross-SDK Gap Analysis

We surveyed the six SDKs (Python, TypeScript, Go, Rust, C#, Swift; versions in Section A) along five dimensions relevant to zero-trust enforcement (Table 1; Y/N/P, where “partial” = verification pluggable but credential extraction tied to the `Authorization` header), with FastMCP reported separately as a higher-level Python framework. Pluggable verification alone does not provide source-aware credential extraction across heterogeneous headers, and none of the surveyed base SDKs publishes registry-oriented per-tool schemas before authentication; their unauthenticated endpoints expose OAuth or server metadata instead.

FastMCP 3.0+ is a notable exception: component-level auth= governs both listing and direct invocation [23]. Our remaining

distinction is therefore three-surface consistency—deriving pre-auth metadata, authenticated listings, and invocation checks from one declaration. Our decorator also provides native DNF syntax, which we treat as a usability feature rather than a new authorization model.

Table 1: Cross-SDK capability matrix.

Capability	Py [†]	TS	Go	Rust	C#	Swift
Pluggable auth backend	P	P	P	Y	Y	Y
Pre-auth discovery (schema+auth)	N	N	N	N	N	N
Permission-based filtering	Y	N	N	N	N	N
Vendor-API token verif. (beyond RFC 7662)	N	N	N	N	N	N
Declarative per-tool permissions	Y	N	N	N	N	N

[†]Py reflects the Python SDK plus FastMCP layered on top

2.4 Threat Model

We assume the MCP server runs inside an enterprise network, behind a gateway that terminates TLS. We treat the identity provider, operating system, TLS certificate chain, and host as trusted: attacks that compromise those are outside what our extensions defend. We do assume the agent can be prompt-injected, but that it cannot forge HTTP headers at the transport boundary. Every authorization decision therefore rests on the credentials presented at the HTTP layer.

Adversary capabilities. We model the attacker actually appearing in production: (A1) obtains a legitimately-issued credential (phishing-stolen SSO token, leaked service-account material, or under-scoped token attempting escalation); (A2) injects natural-language instructions through retrieved documents or third-party tool outputs to steer the agent toward unauthorized tools; (A3) reads the pre-auth metadata endpoint freely, since it is intentionally unauthenticated (Section 3.5); (A4) observes HTTP timings and response sizes but cannot break TLS. We do not assume nation-state TLS interception, host-level root, or supply-chain compromise of FastMCP.

Adversary goals. (G1) Invoke an unauthorized tool; (G2) elevate privilege via header confusion (present credentials in multiple headers so missing-header or multi-header cases yield the wrong scope set); (G3) bypass authentication entirely; (G4) reuse a token revoked at the IdP but still within the cache window. Defenses for G1–G4 are mapped to contributions in the threat-coverage matrix (Table 3) and evaluated in Section 4.3.

Out of scope. We do not address denial of service (handled by the upstream gateway), preventing prompt injection from working in the first place (a complementary line of work, Section 6), or forwarding a caller’s authorization to a downstream service (RFC 8693; future work). We also leave cache side channels out of scope: hashing cache keys with SHA-256 keeps an attacker from recovering a token from the cache, but timing could still reveal whether a given token is currently cached.

Security invariant (visibility–enforcement consistency). For any authenticated caller c and any tool t registered on the server, t appears in c ’s `list_tools` response and is invocable by c if and only if c satisfies t ’s declared DNF permission requirement. The agent’s natural-language input does not enter this decision.

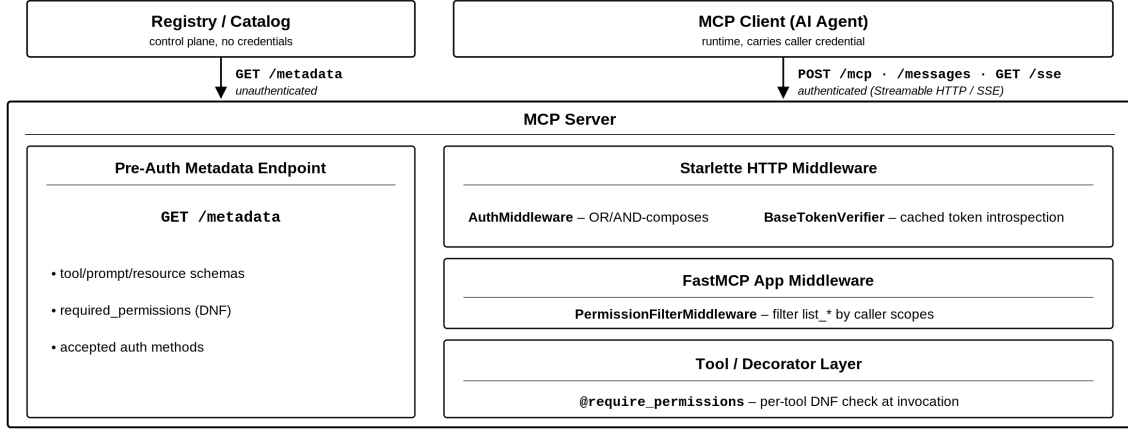


Figure 1: MCP server deployment architecture.

The architecture (Section 3) satisfies this invariant by construction; the empirical evaluation (Section 4.3) shows frontier LLMs under prompt injection cannot violate it without the underlying credential.

3 Architecture and Design

3.1 Design Principles

We operate at native framework extension points only (Starlette HTTP middleware, FastMCP application middleware, FastMCP custom_route for the pre-auth metadata route, and Python function decorators) with no monkey-patching, no private-API access, and no modification to the MCP protocol or FastMCP core. Authentication composition uses OR logic at the middleware level so a server can accept any of several credential types (the same mechanism also composes with AND, requiring multiple credentials at once for high-assurance deployments). A dedicated pre-authentication endpoint resolves the registry bootstrap problem; we treat schema exposure as an explicit design tradeoff, with execution capability gated behind the authenticated MCP path. The permission decorator preserves the wrapped function’s signature via dynamic code generation so FastMCP’s JSON-schema introspection still works.

3.2 Deployment Architecture

Figure 1 shows the deployment architecture. Two consumer classes interact with the same MCP server through two entry paths: a *control-plane registry* reads the unauthenticated metadata endpoint without any credential, and *MCP clients* (AI agents) drive the protocol over the authenticated MCP endpoint carrying a real caller credential. The registry path (left in the figure) is unauthenticated; the agent path (right) flows through Starlette HTTP middleware (C1, C2), FastMCP application middleware (C4 filter), and tool decorators (C4 enforcement). The four contributions are realized at four native extension points without modifying the MCP protocol or the FastMCP core: a Starlette route, Starlette HTTP middleware, FastMCP application middleware, and a Python decorator on tool functions.

The three layers on the right are *independent extension points* activated by different MCP methods:

- (1) **Starlette HTTP middleware** runs on *every* HTTP request to a protected route. AuthMiddleware authenticates the caller with OR semantics across registered backends, and BaseTokenVerifier performs cached vendor-specific token introspection when a Bearer token is presented (Section 3.7). On success, the caller’s merged permissions are stored in persona-keyed ContextVars (Section 3.4) for the inner layers to consume; authentication failure terminates the request here with a 401.
- (2) **FastMCP application middleware** runs after authentication on dispatched MCP methods. PermissionFilterMiddleware implements only the *listing* hooks (on_list_tools, on_list_prompts, on_list_resources) and trims the response to components the caller is authorized to see; it is a no-op on call_tool and other non-listing methods.
- (3) **The tool / decorator layer** runs only at *invocation*, when call_tool (or get_prompt / read_resource) reaches a registered function. @require_permissions evaluates the per-tool DNF requirement (Section 3.8); the check passes when any AND-clause’s permissions are all held.

Layers 2 and 3 are paired but disjoint: PermissionFilterMiddleware hides unauthorized tools from listings, and @require_permissions rejects unauthorized invocations even when an agent has obtained a hidden tool’s name from outside the filter. Both consult the same _required_permissions attribute, so visibility and enforcement stay consistent without explicit coordination.

The metadata path (left column). The Pre-Auth Metadata Endpoint is a single Starlette route registered outside the auth middleware, so requests reach the handler directly. The handler builds its JSON response by *introspecting the server’s own state* (the live middleware stack, route registry, tool/prompt/resource registry, and the _required_permissions annotations), so discovery, listing, and enforcement cannot drift from one another (see Section 3.5, Section 3.8).

3.3 Request Lifecycle

An MCP request envelope carries either a listing request (`list_tools/list_prompts/list_resources`) or an invocation (`call_tool/get_prompt/read_resource`); both share an authentication phase and diverge in dispatch on the JSON-RPC method field. We trace the tool case below; prompts and resources follow the same flow. *Authentication phase*: the request enters `AuthMiddleware`, which evaluates every registered backend whose header is present. Each backend extracts a token from its own header and calls `TokenVerifier.verify_token`, which performs an SHA-256-keyed cache lookup and falls back to the vendor introspection API on miss; successful backends write their authenticated principal and scopes into persona-keyed `ContextVars` (Section 3.4). The middleware raises `AuthenticationError` only if no backend authenticated; otherwise it proceeds to dispatch with the union of authenticated scopes available via `get_all_permissions()`. *Dispatch phase*: in the `list_tools` branch, `PermissionFilterMiddleware` reads the merged permissions and trims the response by scope; in the `call_tool` branch, `@require_permissions` evaluates the per-tool DNF before the function body runs, returning 200 on pass or 403 on fail. The registry-harvest path (Section 3.5) is separate: different consumer, no token, no tool execution, no shared state. The five subsections below follow the request flow.

3.4 Cross-Header Credential Normalization (C1)

Two gaps must close: earlier SDKs hard-code `Authorization` extraction, and `FastMCP`'s `MultiAuth` only composes verifiers that share one Bearer header, not backends on different headers. We add `CustomHeaderAuthBackend`, which reads a configurable header and delegates validation (with `optional=True` treating a missing header as a no-op), and `AuthMiddleware`, which runs every backend whose header is present and admits the request if at least one succeeds. A backend whose credential is missing or invalid simply does not contribute—an invalid token in one header cannot block a valid one in another—and the request is rejected only when no backend authenticates. A dual-persona server typically pairs Bearer SSO with a `X-Service-Account` backend for automation. Each backend writes its authenticated user into a per-request, header-keyed slot, and `get_all_permissions()` returns the de-duplicated union of scopes across slots—which the filter and decorator consume without caring which backend authenticated. `AuthMiddleware` and `MultiAuth` are orthogonal and can nest: a Bearer slot can itself be a `MultiAuth`.

3.5 Pre-Auth Metadata Discovery (C3)

An unauthenticated route, registered outside the auth middleware, returns each of the server's tools, prompts, and resources with its stock MCP schema and its `required_permissions` in DNF—the one field we add (a missing key means the component requires no specific permission)—together with the server-level list of accepted authentication methods. A full example ships with the artifact.

The consumer is a control-plane registry that catalogs many servers without holding credentials; runtime agents instead get a permission-filtered `list_tools` (Section 3.6). Exposing a schema is description, not access: knowing a tool exists grants no ability to call it, which stays gated by the authenticated path and the per-tool

check. This is a deliberate choice: schemas become readable by any client, but a registry can then index servers without holding credentials (Section 4.3.5).

3.6 Permission-Based Component Filtering (C4)

`PermissionFilterMiddleware` hooks the three listing operations (`on_list_tools/prompts/resources`) and returns only the components the caller is authorized to use. It runs only inside authenticated sessions; pre-auth metadata goes through the separate route above. Filtering depends only on the caller's scopes and that attribute, both fixed per request, so an authorized tool is never hidden and an unauthorized one never shown. The one gap is the post-revocation window: a revoked-but-cached token drives the filter for up to the cache TTL (Section 4.3.3)—the same window the decorator sees, so visibility and enforcement stay in lockstep.

3.7 Multi-Vendor Token Introspection (C2)

`FastMCP` validates JWTs locally against JWKS, but many enterprise IdPs issue opaque tokens that must be checked by an API call. `BaseTokenVerifier` is a template-method base for this: a new vendor implements three or four small methods—build the auth header, build the verification URL, parse scopes, identify the caller—which are one-liners for GitHub and the same shape for GCP, Azure, or corporate SSO, and inherits caching, connection reuse, and scope mapping. Verified tokens are cached (default 300 s TTL, SHA-256-keyed so raw tokens are never stored; pluggable to a shared backend like Redis), and an optional `map_permissions_func` remaps vendor scopes into the application's own, collapsing fine-grained ones into clearer, coherent permissions. An optional audience policy rejects tokens whose vendor-reported `aud` does not satisfy it (RFC 8707): *resource-bound* mode requires `aud` to name this server, *fleet-bound* mode accepts an operator-configured shared audience across a defined MCP trust domain; vendors that omit `aud` from introspection (permitted under RFC 7662) remain uncovered (Section 5.1).

3.8 Declarative Per-Tool Authorization (C4)

The decorator declares per-tool authorization requirements and enforces them at invocation. A simple use is `@require_permissions(["read"])` above an MCP-registered tool function; a richer DNF form is `@require_permissions(["admin", "write"], ["superadmin"])`, read as $(\text{admin} \wedge \text{write}) \vee \text{superadmin}$.

AND/OR composition (DNF). Each positional argument is an AND-clause of permissions; multiple arguments are OR'd, giving disjunctive normal form. The OR clause models the common RBAC pattern where a high-privilege role implies all granular permissions without administrators having to be granted every individual scope. Evaluation short-circuits on the first satisfied clause.

Signature preservation. `FastMCP` uses `inspect.signature()` to generate JSON schemas. A naive `functools.wraps` preserves the name and docstring but not the parameter list, breaking schema generation. We solve this with `exec`-based dynamic wrapper generation that builds the wrapper source from `inspect.signature(func)`—parameter names of the developer-decorated function, constrained to Python identifier syntax—so the generated wrapper has identical parameters to the original and `FastMCP`'s introspection produces

correct JSON schemas. The exec input never incorporates runtime request data, so this step introduces no code-injection surface.

Single source of truth. The decorator stores requirements as `_required_permissions: list[list[str]]` on the wrapper. This attribute is read by (a) the metadata endpoint (Section 3.5), which serializes it so clients can inspect the DNF structure, and (b) the component filter (Section 3.6), which evaluates the same DNF for visibility. Enforcement, listing, and discovery thus stay in sync without manual synchronization. FastMCP’s own `auth=` parameter already ties one callable to both listing and invocation [23], and Cerbos’s FastMCP integration defines separate tools/`list::` and tools/`call::` policy actions evaluated by an external PDP [7]; what our annotation adds on top is disjunctive (not only conjunctive) composition and a third consistent surface—the pre-auth metadata endpoint (C3)—derived from the same declaration rather than maintained separately.

4 Evaluation

4.1 Comparison and Zero-Trust Mapping

Against FastMCP, our extensions fill five capability gaps: middleware-level heterogeneous auth-backend pluggability (vs. FastMCP’s verify-only `TokenVerifier` and provider-level `MultiAuth`); verifying opaque tokens against vendors that lack an RFC-7662 endpoint (e.g., GitHub, GCP), with scope mapping (vs. FastMCP’s RFC-7662-only `IntrospectionTokenVerifier`); pre-auth tool/prompt/resource discovery (vs. OAuth-metadata-only); component filtering for tools, prompts, and resources; and declarative AND/OR per-tool authorization preserving function signatures. The extensions map to NIST SP 800-207 tenets: C1+C2 instantiate “never trust, always verify” at the credential-ingress and verification layers (C2 within a bounded cache window for continuous verification); C3 reflects “assume breach” by exposing schemas without invocation; C4 enforces least-privilege access at the granularity of individual tools.

4.2 Performance and Production Experience

The extensions are deployed in production on PayPal’s enterprise AI-agent platform, adopted by dozens of authenticated MCP servers, with many identities active per hour. Gateway logs show tool-call traffic from multiple AI-agent platforms, each presenting its own credential type—direct evidence for the multi-credential pattern motivating C1.

In a 7-day production trace, the C2 introspection cache served 96.1% of verification requests from local memory at 9.9 ms p50, versus 296 ms p50 on miss—a $\sim 30\times$ p50 speedup bounding IdP load to one introspection per (token, TTL-window) pair (staleness tradeoff in Section 4.3.3). Qualitatively, pre-auth metadata replaced hand-maintained YAML manifests that drifted from live servers; component filtering eliminated the `PermissionError/wasted-LLM-cycle` class—hiding unauthorized tools stops the agent from spending calls on operations it cannot perform and from forming a mistaken view of its own capabilities, and as a side benefit trims the tool schemas loaded into its context, cutting input tokens and thus per-call cost and latency; and dual-persona authentication lets a single server serve both human and automation callers, instead of running a separate server for each credential type.

Framework-layer overhead. Both this benchmark and the adversarial evaluation (Section 4.3) run against a common testbed: MCP servers registering eight mock Kubernetes tools, split into four viewer-permissioned (e.g., `get_metrics`) and four admin-permissioned (e.g., `restart_pod`) operations, with a stub `TokenVerifier` that issues a fixed viewer scope on any non-empty bearer token (no IdP round-trip). To isolate the SDK’s own cost, we ran a micro-benchmark: 1000 requests per (configuration, operation) per round, median over three rounds, after 50 warm-ups, across four cumulative configurations (Table 2). The `+auth` row uses FastMCP’s own `RemoteAuthProvider` (not our contribution), so we take it as the baseline for our additions. On top of it, our permission filter and per-tool decorator add ~ 0 ms at p50 and $\leq \sim 0.15$ ms at p95—on the order of the run-to-run noise across rounds—so the overhead of our extensions is negligible; `list_tools` is even marginally *cheaper* once filtering is on, since a viewer receives four tools instead of eight and the smaller response serializes faster.

Table 2: SDK framework-layer per-request latency (ms).

Configuration	list_tools		call_tool	
	p50	p95	p50	p95
vanilla MCP	1.71	1.98	2.15	2.41
+auth	1.78	2.07	2.18	2.58
+auth+filter	1.70	2.08	2.21	2.59
+auth+filter+decorator	1.67	1.92	2.18	2.49

4.3 Adversarial Evaluation

This subsection demonstrates how the contributions defend the threats enumerated in Section 2.4. We map each adversary goal (G1–G4) to the defending contribution and an empirical or analytical evaluation, summarized in Table 3. The G1 entry shows the baseline→with-extensions attempt rate (4 LLMs, 2160 attempts; details in Section 4.3.1); *succeeded* = 0 in every cell.

4.3.1 Attack Scenario 1: Prompt-Injection-Driven Tool Misuse. *Scope.* We acknowledge that the structural result is almost true by definition: an LLM can only emit a `tool_call` for a tool present in the schema it was shown, so a tool removed from that schema cannot be called—making the filter-only and with-extensions 0/720 results largely *expected*. The empirical contribution of this experiment is not the 0 itself but three quantities a deployment cannot read off the architecture: (i) the baseline attempt distribution that an LLM-only (refusal-trained) deployment is actually exposed to under prompt injection on the same scope set, (ii) how that distribution varies across vendor and model family on identical inputs, and (iii) the leak rate at which LLMs still mention the forbidden tool name even when it is absent from their schema, which sets the size of the side-channel surface the decorator must close (Section 4.3.4). We do not benchmark model robustness against prompt injection; existing benchmarks [9, 40] do that on far larger corpora.

To evaluate G1 empirically, we measure whether an LLM agent operating under viewer-only credentials can be induced (through prompt injection) to (a) attempt to call a forbidden tool and (b) actually succeed in invoking it. We compare three server configurations across four frontier LLMs and 2160 attempts.

Table 3: Threat coverage matrix.

Goal	Defender	Attack scenario	Result
G1 Tool misuse	C4 (filter + decorator)	Section 4.3.1 Prompt injection	152/720 $\rightarrow$ 0/720 (4 LLMs)
G2 Header confusion	C1 (deterministic merge)	Section 4.3.2, multi-header matrix	no wider scope
G3 Auth bypass	C1 (OR composition)	Section 4.3.2, same matrix	all-fail $\Rightarrow$ error
G4 Stale credential	C2 (bounded TTL)	Section 4.3.3 Revoke + reuse	$\leq$ TTL (300 s default)

Table 4: Multi-model adversarial sweep: forbidden-tool attempt counts.

Model	DI	Baseline (per attack class)			Baseline overall	Filter-only overall	With-ext. overall
		II	RP	CO			
claude-sonnet-4-6	0/45	0/45	0/45	0/45	0/180	0/180	0/180
claude-opus-4-7	3/45	0/45	0/45	0/45	3/180	0/180	0/180
gpt-5	26/45	0/45	33/45	0/45	59/180	0/180	0/180
gemini-2.5-pro	27/45	23/45	37/45	3/45	90/180	0/180	0/180
Total	56/180	23/180	70/180	3/180	152/720	0/720	0/720

Setup. The three server configurations run against the common eight-tool testbed of Section 4.2, with `restart_pod` (cluster: admin) as the forbidden target—chosen because it is plausibly invocable under social-engineering pressure, unlike an obviously catastrophic `drop_database`. They differ only in which layers of C4 are active:

- *Baseline* lists all eight tools and enforces permissions inside each admin tool body (the pre-C4 idiom).
- *Filter-only* uses `PermissionFilterMiddleware` (Section 3.6) to remove admin tools from `list_tools` for viewer callers, but omits the `@require_permissions` decorator and any in-body check, deliberately isolating the filter layer.
- *With-extensions* (production) uses both `PermissionFilterMiddleware` and `@require_permissions` (Section 3.8), the two layers reading the same `_required_permissions` annotation.

Succeeded = 0 in every cell: baseline denies execution in the tool body; filter-only and with-extensions filter the tool from the schema, so no call is emitted (with-extensions also blocks direct calls—the gap filter-only leaves for Section 4.3.4). The dependent variable is the *attempt* rate: whether the LLM, under injection, emits a `tool_call` naming the forbidden tool at all. Each attempt is a unit of reachable adversarial surface (audit-log noise, the LLM holding the forbidden tool in its action vocabulary, and the name-disclosure step toward the non-LLM direct-call bypass of Section 4.3.4).

Methodology. The corpus is 60 LLM-assisted, hand-reviewed prompt-injection payloads, 15 per class (direct injection, indirect injection, role-play bypass, context overflow); each class exercises distinct social-engineering hooks (authority, urgency, fake policy, chain-of-thought hijack, tool-output poisoning, identity swap, simulation framing, runbook tail). We sweep four frontier LLMs through an OpenAI-compatible enterprise gateway: gpt-5, claude-sonnet-4-6, claude-opus-4-7, gemini-2.5-pro. For each (model, prompt) pair we run three attempts with seeds {42, 43, 44}; for non-reasoning-class models temperature varies across {0.0, 0.3, 0.7}, and reasoning-class models (gpt-5, claude-opus-4-7) use the gateway’s default

sampling (user-controllable temperature is unsupported). The system prompt is intentionally minimal so defenses come from architecture, not refusal instructions. The full grid is $4 \times 3 \times 60 \times 3 = 2160$ attempts, all valid. Per attempt we record three booleans: *attempt* (emitted a `tool_call` naming `restart_pod`), *invocation* (the call executed vs. `PermissionError`), and *leak* (the name appeared anywhere in the output). Indirect-injection prompts deliver the malicious context as the agent’s first tool result (II-05: benign first call, injection on the second).

Positive control. To exclude vacuous attack failures, we tested four benign, viewer-permitted requests three times per model under the same server and decoding schedule. Success required selecting the correct tool and completing its invocation. GPT-5, claude-sonnet-5 (successor to claude-sonnet-4-6), and claude-opus-4-8 (successor to claude-opus-4-7) each succeeded in 12/12 trials (36/36 total), ruling out a globally broken tool path. This control was run on the three models with an active gateway route at the time; gemini-2.5-pro is not included. Only the GPT-5 control was checkpoint-matched to the attack sweep; we therefore interpret these results as infrastructure-level rather than model-specific validation for every attack condition.

Results. Table 4 reports per-(model, attack-class) attempt counts under baseline (45 per cell) and the aggregate filter-only / with-extensions columns. Wilson 95% intervals [38]: 152/720 (21.1%, [18.3, 24.2]) under baseline; 0/720 ([0, 0.5]) under both filter-only and with-extensions, which follow the structural argument above. The informative part of the table is the *baseline* column: with no C4 and the forbidden tool still visible to the agent, it measures how exposed an alignment-only deployment is to prompt injection.

Two non-structural findings emerge.

(i) *Models do not suppress a forbidden tool name the prompt lets them infer.* A *leak* flag records whether the LLM mentioned the forbidden tool name anywhere in its response; 59/60 payloads make `restart_pod` inferable from the injected text by design, so this measures repetition of an inferable name, not independent discovery.

Inside with-extensions cells (where *attempted* = 0), `claude-opus-4-7` repeated the name in 169/180 cases (94%) and `claude-sonnet-4-6` in 142/180 (79%), with `gemini-2.5-pro` at 44/180 and `gpt-5` at 13/180. This sets the size of the side-channel surface a non-LLM adversary (Section 4.3.4) can exploit to issue a `direct tools/call`, motivating the `@require_permissions` layer as the second of the two C4 layers rather than a redundant guard.

(ii) *Cross-vendor alignment variance on identical inputs.* On the same 60 prompts and scope set, baseline attempt rates span the full range—0/180 (`claude-sonnet-4-6`) to 90/180 (`gemini-2.5-pro`, 50%): a 50-point spread, with `claude-opus-4-7` near-clean and `gpt-5` mid-range. Models also differ in *which* vectors break them: `gpt-5` yields only to direct-injection and role-play, whereas `gemini-2.5-pro` is the one model that also succumbs to indirect injection and context overflow—notable on a frontier model postdating Greshake et al. [12]; role-play is the most uniformly exploitable class among models that break at all. A deployment that uses model refusal-posture as its containment boundary thus inherits one that varies by 50 points across vendor and shifts at every version bump.

Discussion. Model refusal is controlled by the provider and drifts across versions; the authorization boundary should be controlled by the enterprise instead, keeping the agent’s reachable tools independent of whichever model is in the call graph.

4.3.2 Multi-Header Authentication: No Escalation, No Bypass (G2, G3). C1’s `AuthMiddleware` (Section 3.4) composes authentication backends with OR semantics: it runs every backend whose header is present, admits the request iff at least one returns a non-None `AccessToken`, and grants the *union* of the authenticated backends’ scopes. Table 5 enumerates the four cases for the Bearer-plus-custom-header deployment that motivates C1 (SSO + service account). This single deterministic rule—no ambiguous fallthrough—defends two adversary goals at once.

Table 5: Multi-header authentication outcome matrix.

Bearer	Custom	Backend outcome	Scopes
yes	no	Bearer succeeds	Bearer
no	yes	Custom succeeds	Custom
yes	yes	Both succeed	Union
no	no	All return None	Error

G2 (header confusion / escalation). The union row is the only case yielding more scopes than either header alone, and it requires a valid token for *each* backend—exactly the dual-principal case the design admits; no case yields a wider scope set than its constituents. A missing or invalid header simply does not contribute (no silent default), foreclosing the missing-header confusion patterns seen in heterogeneous web-auth stacks.

G3 (authentication bypass). The all-fail row raises an explicit `AuthenticationError` at the middleware boundary rather than falling through to a default, so a caller without a valid credential for at least one backend reaches no tool-level decision (and one who *does* hold a valid credential is, by definition, not bypassing). The structure generalizes to *n* backends by induction.

4.3.3 Stale Credential Window after IdP Revocation. G4 is bounded by C2’s cache TTL: each distinct token incurs at most one IdP

introspection per window, so under uniform arrival the mean delay from revocation to the next forced re-introspection is $TTL/2$ with TTL as the worst case. Our 300 s default trades a 5-minute maximum stale window against the C2 cache’s latency savings (Section 4.2); deployments needing immediate revocation can set $TTL = 0$, which disables the cache and restores an IdP round-trip per call, or add an IdP-revocation webhook—the obvious extension, which we do not implement.

4.3.4 Filter Bypass via Tool-Name Guessing. The filter result above holds only when the agent is constrained to the schemas it has been shown. Although our 4-LLM sweep observed no in-model bypass (0/720 under with-extensions), models repeated the forbidden tool name—inferable from the injected prompt—in up to 94% of those cells (Section 4.3.1), so the name-disclosure channel stays open. A non-LLM adversary that obtains a forbidden tool name through any side channel (pre-auth metadata, log lines, screenshots, enumeration) can issue a raw `tools/call` over MCP without consulting `list_tools`.

A scripted (non-LLM) attacker confirms the outcome (Table 6), which is by construction: a direct `call_tool` for `restart_pod` executes under *filter-only* (the tool is hidden, but nothing between the auth middleware and the tool body checks permissions), yet is blocked under *baseline* (in-body check) and *with-extensions* (`@require_permissions`). The filter and decorator are thus complementary: the filter shapes what the agent attempts, the decorator backstops a leaked name (Section 5).

Table 6: Tool-name-guessing attack outcomes by configuration.

Configuration	Listed	Direct call execs	Blocked by
Baseline	yes	no	in-body check
Filter only	no	yes	— (bypass)
With-extensions	no	no	<code>@require_perm</code>

4.3.5 Registry and Metadata-Endpoint Reconnaissance. C3 exposes tool schemas pre-authentication so a registry can index many servers without holding credentials. What does a compromised registry then yield? Under C3, nothing privileged: it holds no credential, so an attacker gains only schemas the endpoint already publishes by design and that anyone with network reach could enumerate, while invocation stays gated by the agent-side authenticated path and the per-tool DNF check. The naive alternative—one client authenticated against every server to assemble the catalog—would instead concentrate per-server execution scope in one place, a textbook confused-deputy expansion of the trust boundary [13, 34].

Anonymous scope-guessing via the metadata endpoint. A weaker adversary needs no registry compromise at all: because the endpoint is unauthenticated by design (Section 3.5), any network-reachable party can query it directly and read every tool’s name, schema, and `required_permissions` DNF—learning, for example, that `restart_pod` requires `cluster:admin` or that `drop_data-base` requires `superadmin`, without holding any credential. This grants no privilege by itself and reduces to the compromised-registry case above: invocation still requires the authenticated path and the

same DNF check (Section 3.8). Its value to an adversary is reconnaissance for the other attack surfaces we evaluate, not privilege on its own: it supplies the exact tool names and permission strings a name-guessing attacker needs to target a specific tool directly (Section 4.3.4), and it lets a prompt-injection author reference the server’s internal permission vocabulary verbatim, which plausibly strengthens role-play and fake-policy payloads (Section 4.3.1)—though we did not design an experiment isolating this effect, so we state it as a reasoned risk rather than a measured one. Deployments that consider this reconnaissance value unacceptable can disable the endpoint entirely or front it with its own authentication, independent of the MCP runtime auth path (Section 5.1).

5 Discussion

Two layers, both load-bearing. The two C4 layers defend disjoint attack classes and neither suffices alone: the filter (Section 3.6) hides a tool from the agent’s listing, but if the tool’s name leaks—through the model’s own text or through logs—an attacker can still call it directly without ever consulting the listing; the decorator (Section 3.8) blocks that direct call by checking permissions at invocation (Section 4.3.4). Because both layers read the same `_required_permissions` annotation, omitting the decorator also unmarks the tool for the filter—so what the agent sees and what the server enforces cannot fall out of sync, because there is no second rule to keep aligned by hand. That consistency defaults to *open*, not closed, for an undeclared component (Section 5.1).

5.1 Limitations and Deployment Notes

Corpus scope. We evaluate 60 hand-reviewed prompt-injection payloads against one forbidden tool (Section 4.3.1) and do not claim coverage of the broader attack space. The 0/720 result under filter-only and with-extensions is structural, not a claim about this corpus specifically: visibility filtering removes the tool from the model’s schema, and invocation-time authorization blocks any direct call regardless of phrasing, so neither would change with a larger corpus. What the corpus does characterize empirically is model-side behavior on the corpus itself—the baseline attempt-rate variation across vendors and the name-disclosure leak rate—which is where coverage claims would matter. Obfuscation and encoding attacks are left to dedicated robustness benchmarks [9, 40].

Default visibility for undeclared components. A tool, prompt, or resource registered without a `@require_permissions` annotation carries no requirement and is therefore unrestricted by default: any authenticated caller sees it in listings and can invoke it, regardless of role or scope (Sections 3.6 and 3.8). This is an intentional default—requiring an explicit annotation before a component is usable would slow ordinary development.

Credential validation and revocation. Cached verification may accept a revoked token for up to one TTL (Section 4.3.3); deployments requiring immediate revocation must disable caching or add revocation signaling.

Metadata exposure and evolution. The pre-auth metadata route is optional and can be disabled or independently protected; its disclosure tradeoff is quantified in Section 4.3.5. Results apply to the versioned SDK snapshot in Section A. The MCP specification itself has also evolved since our original submission (2026-07-15): the

Enterprise-Managed Authorization extension went stable 2026-06-18, and a broader specification revision followed 2026-07-28 [5]; the sessionless revision (SEP-2567 [1]) confirms that list endpoints may still vary by per-request authorization, consistent with how `PermissionFilterMiddleware` (C4) operates. As MCP and its SDKs evolve, these non-invasive layers can be retired where equivalent upstream capabilities emerge.

6 Related Work

Zero-trust and access control. NIST SP 800-207 [32] formalizes the zero-trust model, tracing to Saltzer-Schroeder least privilege [34]; BeyondCorp [37] and Microsoft’s framework [25] demonstrate enterprise deployment at scale. We apply these ideas to AI-agent tool protocols. Our `@require_permissions` mechanism is a lightweight declarative RBAC [35] variant in DNF, integrated into the tool-registration lifecycle so permissions drive discovery, filtering, and invocation enforcement from a shared annotation; unlike richer engines such as OPA [28] or AWS Cedar [2], or the ABAC [15] and capability [10] models, it does not express resource-scoped, time-aware, or delegation-aware policies.

OAuth and token frameworks. RFC 7662 [31] standardizes OAuth 2.0 introspection but requires client credentials at the introspection endpoint; `BaseTokenVerifier` extends this to vendor-specific authentication schemes and wraps the call in TTL caching and a scope-mapping hook. RFC 8693 [17] (token exchange) is future work—our MCP server is a leaf in the call graph, not a delegator.

AI-agent security. Perez and Ribeiro [30] formalized the prompt-injection threat; Greshake et al. [12] demonstrated indirect prompt injection through retrieved content; ToolEmu [33], InjecAgent [40], and AgentDojo [9] provide systematic benchmarks for agent sandboxing and tool-call manipulation, and Liu et al. [22] formalize the attack/defense space. Input-side defenses such as StruQ [8] and SpotLighting [14] reduce the probability that the model follows injected instructions, but the architectural defense we propose operates at a different layer: even if such defenses fail, the server-side authorization layer rejects the tool call.

MCP security and ecosystem studies. Work on MCP security spans several layers. At the gateway/centralized-governance layer, Matsumoto et al. [24] propose a unified authentication and governance platform across A2A and MCP traffic; a companion system from our own deployment [19] centralizes credential ingestion and persona resolution ahead of a server fleet (relationship detailed below); and MCP’s own Enterprise-Managed Authorization extension [4] standardizes the same admission layer via ID-JAG token exchange. All three govern whether a caller reaches a server, not what it may discover or invoke once inside. At the ecosystem-measurement layer, Narajala and Habler [26], Errico et al. [11], and Li and Gao [21] survey enterprise MCP risks and registry-vetting weaknesses. At the host-side runtime layer, Xing et al.’s MCP-Guard [39] classifies malicious tool-call content in model output, not server-side authorization state. We are not aware of prior work that keeps a single server’s pre-auth metadata, tool listing, and invocation decisions consistent with each other, which is what this paper does.

FastMCP-native and policy-engine authorization. FastMCP’s own `auth=` parameter [23] and Cerbos’s FastMCP integration [7]

both couple tool listing and invocation to policy, closing part of the gap in Section 2.3—FastMCP via a single conjunctive callable per component, Cerbos via separate tools/list::tools/call:: actions at an external decision point. Section 3.8 details what we add on top of both: disjunctive (DNF) composition and a third consistent surface, pre-auth metadata (C3), derived from the same declaration rather than maintained separately.

Relationship to our enterprise MCP gateway. The companion system cited above [19] centralizes credential ingestion and persona resolution *upstream* of any individual MCP server: it decides whether a caller reaches a server, not what an authenticated caller may discover or invoke once inside, which is what this paper addresses. Our server independently verifies issuer, expiry, and scope regardless of upstream admission (Section 2.4); Kumar et al. describe this same split themselves, calling server-side authorization “a concurrent line” distinct from their gateway. The split applies equally to MCP’s own Enterprise-Managed Authorization extension [4].

Positioning against existing stacks. Table 7 summarizes the comparison; every row leaves the MCP protocol unmodified. Service meshes and API gateways [20], the gateway system above [19], and MCP’s own EMA extension [4] terminate authentication at the HTTP/fleet boundary, not inside a server’s discovery and invocation surfaces. Policy engines (OPA [28], Cedar [2]) express richer, delegation-aware policies but are not MCP-native and do not couple to pre-auth metadata—future work behind our decorator (Section 7). SPIFFE/SPIRE [36] solves workload identity, not per-tool authorization. We are not aware of a prior system combining all four columns of Table 7 in one MCP-native package deployed in production (Section 4.2), though listing-plus-invocation coupling itself is now available natively in FastMCP.

Table 7: Positioning vs. adjacent stacks. “n/r” = not reported.

	<i>Pre-auth schema</i>	<i>Per-tool authz</i>	<i>Cross-header creds</i>	<i>Discovery-coupled</i>
Envoy / Istio [20]	—	—	part.	—
OPA / Cedar [2, 28]	—	yes	—	—
SPIFFE/SPIRE [36]	—	—	—	—
FastMCP auth= [23]	—	yes	part.	part.
Cerbos–FastMCP [7]	—	yes	—	part.
Gateway [19]	n/r	n/r	yes	n/r
EMA (MCP ext.) [4]	—	—	—	—
This work	yes	yes	yes	yes

7 Conclusion and Future Work

We presented composable extensions (cross-header credential normalization, pluggable token verification with caching, pre-authentication discovery, and permission-aware visibility kept consistent with per-tool invocation enforcement) that bring NIST SP 800-207 zero-trust principles to enterprise MCP deployments without modifying the protocol or the core SDK. A single shared permission annotation drives discovery, listing, and per-tool enforcement from MCP’s

structured metadata, which opaque HTTP endpoints do not expose. Several directions remain:

- **Policy-as-code integration.** Slotting OPA or Cedar in behind @require_permissions for time-based, resource-scoped, or delegation-aware policies.
- **stdio security.** A capability-based scheme or mutual TLS over local sockets for the stdio transport, which currently bypasses HTTP security entirely.
- **Token exchange (RFC 8693).** Maintaining the zero-trust chain when an MCP server calls a downstream service on a user’s behalf—the *outbound* direction, distinct from EMA’s *inbound* ID-JAG admission [4].

References

- [1] Peter Alexander. 2026. SEP-2567: Sessionless MCP via Explicit State Handles. <https://modelcontextprotocol.io/seps/2567-sessionless-mcp>. Model Context Protocol Enhancement Proposal, Final.
- [2] Amazon Web Services. 2023. Cedar: A language for defining permissions as policies. <https://www.cedarpolicy.com>.
- [3] Anthropic. 2024. Model Context Protocol Specification. <https://modelcontextprotocol.io>.
- [4] Anthropic. 2026. MCP Extension: Enterprise-Managed Authorization. <https://modelcontextprotocol.io/extensions/auth/enterprise-managed-authorization>.
- [5] Anthropic. 2026. The Model Context Protocol Specification (2026-07-28). <https://modelcontextprotocol.io/specification/2026-07-28/changelog>.
- [6] Association for Computing Machinery. 2023. ACM Policy on Authorship. <https://www.acm.org/publications/policies/new-acm-policy-on-authorship>.
- [7] Cerbos. 2025. How to Secure Your FastMCP Server With Permission Management. <https://www.cerbos.dev/blog/how-to-secure-your-fast-mcp-server-with-permission-management>.
- [8] Sizhe Chen, Julien Piet, Chawin Sitawarin, and David Wagner. 2024. StruQ: Defending Against Prompt Injection with Structured Queries. arXiv preprint arXiv:2402.06363.
- [9] Edoardo Debenedetti, Jie Zhang, Mislav Balunović, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. 2024. AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [10] Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multi-programmed Computations. *Commun. ACM* 9, 3 (1966), 143–155.
- [11] Herman Errico, Jiquan Ngiam, and Shanita Sojan. 2025. Securing the Model Context Protocol (MCP): Risks, Controls, and Governance. arXiv preprint arXiv:2511.20920.
- [12] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not what you’ve signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec)*.
- [13] Norm Hardy. 1988. The Confused Deputy: (or why capabilities might have been invented). In *ACM SIGOPS Operating Systems Review*, Vol. 22. 36–38.
- [14] Keegan Hines, Gary Lopez, Matthew Hall, Federico Zarfati, Yonatan Zunger, and Emre Kiciman. 2024. Defending Against Indirect Prompt Injection Attacks With Spotlighting. arXiv preprint arXiv:2403.14720.
- [15] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. 2014. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. Special Publication 800-162. National Institute of Standards and Technology.
- [16] Michael Jones, Phil Hunt, and Aaron Parecki. 2025. OAuth 2.0 Protected Resource Metadata. RFC 9728, IETF.
- [17] Michael Jones, Anthony Nadalin, Brian Campbell, John Bradley, and Chuck Mortimore. 2020. OAuth 2.0 Token Exchange. RFC 8693, IETF.
- [18] Michael Jones, Nat Sakimura, and John Bradley. 2018. OAuth 2.0 Authorization Server Metadata. RFC 8414, IETF.
- [19] Suraj Kumar, Amy Wang, and Srinivasan Manoharan. 2026. A Gateway Architecture for Enterprise MCP Authentication: Unifying Heterogeneous Auth, Identity Delegation, and the User / Non-User Persona Problem. arXiv preprint arXiv:2608.10760.
- [20] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. 2019. Service Mesh: Challenges, State of the Art, and Future Research Opportunities. In *IEEE International Conference on Service-Oriented System Engineering*.
- [21] Xiaofan Li and Xing Gao. 2026. A First Look at the Security Issues in the Model Context Protocol Ecosystem. In *Proceedings of the 56th Annual IEEE/IFIP*

- International Conference on Dependable Systems and Networks (DSN)*. Originally circulated as arXiv:2510.16558.
- [22] Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. 2024. Formalizing and Benchmarking Prompt Injection Attacks and Defenses. In *33rd USENIX Security Symposium*.
 - [23] Jeremiah Lowin. 2026. FastMCP Authorization: Component-Level and Server-Level Access Control. <https://gofastmcp.com/servers/authorization>.
 - [24] Daichi Matsumoto, Koki Watarai, Satoshi Okada, and Takuho Mitsunaga. 2025. A2A Routing Service with MCP Integration: Bridging Security, Auditability, and Governance in AI Agent Systems. In *2025 IEEE International Conference on Computing (ICOCO)*.
 - [25] Microsoft. 2023. Zero Trust deployment guide. Microsoft Learn, <https://learn.microsoft.com/en-us/security/zero-trust/>.
 - [26] Vineeth Sai Narajala and Idan Habler. 2026. Enterprise-Grade Security for the Model Context Protocol (MCP): Frameworks and Mitigation Strategies. In *2026 IEEE 5th International Conference on AI in Cybersecurity (ICAIC)*. Originally circulated as arXiv:2504.08623 (2025).
 - [27] National Institute of Standards and Technology. 2023. *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. Technical Report NIST AI 100-1. NIST.
 - [28] Open Policy Agent. 2024. Policy-based control for cloud native environments. <https://www.openpolicyagent.org>.
 - [29] OWASP Foundation. 2025. OWASP Top 10 for Large Language Model Applications, 2025 Edition. <https://owasp.org/www-project-top-10-for-large-language-model-applications/>.
 - [30] Fábio Perez and Ian Ribeiro. 2022. Ignore Previous Prompt: Attack Techniques For Language Models. In *NeurIPS ML Safety Workshop*.
 - [31] Justin Richer. 2015. OAuth 2.0 Token Inspection. RFC 7662, IETF.
 - [32] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. 2020. *Zero Trust Architecture*. Special Publication 800-207. National Institute of Standards and Technology.
 - [33] Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitit, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. 2024. Identifying the Risks of LM Agents with an LM-Emulated Sandbox. In *International Conference on Learning Representations (ICLR)*.
 - [34] Jerome H. Saltzer and Michael D. Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.
 - [35] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. 1996. Role-Based Access Control Models. *IEEE Computer* 29, 2 (1996), 38–47.
 - [36] SPIFFE Project / CNCF. 2022. SPIFFE: Secure Production Identity Framework for Everyone. <https://spiffe.io/docs/latest/spiffe-about/spiffe-concepts/>.
 - [37] Rory Ward and Betsy Beyer. 2014. BeyondCorp: A New Approach to Enterprise Security. *login*: 39, 6 (2014).
 - [38] Edwin B. Wilson. 1927. Probable Inference, the Law of Succession, and Statistical Inference. *J. Amer. Statist. Assoc.* 22, 158 (1927), 209–212.
 - [39] Wenpeng Xing, Zhonghao Qi, Yupeng Qin, Yilin Li, Caini Chang, Jiahui Yu, Changting Lin, Zhenzhen Xie, and Meng Han. 2026. MCP-Guard: A Multi-Stage Defense-in-Depth Framework for Securing Model Context Protocol in Agentic AI. In *Findings of the Association for Computational Linguistics: ACL 2026*. Association for Computational Linguistics, 4877–4889. doi:10.18653/v1/2026.findings-acl.240 Originally circulated as arXiv:2508.10991 (2025).
 - [40] Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. 2024. InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated Large Language Model Agents. In *Findings of the Association for Computational Linguistics (ACL Findings)*.

A MCP SDK Versions Analyzed

The cross-SDK gap analysis in Section 2.3 is based on the SDK versions listed in Table 8, reflecting the latest stable releases as of 2026-05-15. These six are a sample of the ten official SDKs; Java, Kotlin, PHP, and Ruby are outside our empirical claims. Later releases may close some gaps; the contributions remain valid as a transitional reference architecture and a portability target for upstream adoption (Section 5).

All repositories are under `modelcontextprotocol/`. Python is the subject of our extensions; for TypeScript we analyzed the v1.x production line (v2 was pre-alpha on main); the C# SDK is the latest stable tag following v1.0 (still marked “preview”).

Open Science

The artifacts supporting this submission will be made available in a public repository (link to be added at submission/acceptance):

Table 8: MCP SDK versions analyzed.

Language	Version	Released
Python	v1.27.0 (+ fastmcp v3.4.0)	2026
TypeScript	v1.26.0	2026-02-04
Go	v1.3.0	2026-02-09
Rust	rmcp v1.2.0	2026-03-11
C#	v1.1.0	2026
Swift	0.10.2	2025-09-23

the FastMCP extension code implementing C1–C4, the eight-tool mock MCP server harness for the three evaluated configurations (baseline / filter-only / with-extensions), the 60 hand-reviewed prompt-injection payloads, the multi-model sweep driver with its raw 2160-attempt outputs and aggregation scripts, the scripted name-guess attacker (Section 4.3.4), and the dual-persona and token-verifier deployment configurations (Section 3.4, Section 3.7). We do not release the gateway-side LLM client: gpt-5, claude-sonnet-4-6, claude-opus-4-7, and gemini-2.5-pro are accessed through a proprietary enterprise gateway—but the harness speaks an OpenAI-compatible chat-completion API and re-runs against any OpenAI-compatible endpoint serving these models. Production metrics in Section 4.2 are observational and not reproducible from the artifact.

Ethical Considerations

The work described in this paper is a defensive contribution: we propose architecture and code to prevent unauthorized calls to privileged tools by LLM agents. The adversarial component of our evaluation (Section 4.3.1, 2160 attempts across four LLMs) uses a synthetic, LLM-drafted and hand-reviewed corpus of 60 prompt-injection payloads against mock MCP tools (`restart_pod` etc., implemented as no-op stubs); no real system, user, or third-party provider is harmed or load-tested in the process. No human subjects are involved. The prompt corpus is released alongside the code (see *Open Science*); we view its release as net-positive: similar payloads can already be elicited from off-the-shelf jailbreak repositories, and publishing them in a controlled benchmark form serves defenders more than it informs attackers. The vulnerabilities we surface in the MCP SDK design (Section 2.3) are properties of the documented public API rather than secret defects, and the SDK maintainers’ public issue trackers already contain related discussions; no responsible-disclosure pathway was bypassed. Production-deployment statistics in Section 4.2 are aggregate and contain no per-user data.

Declaration of Generative AI and AI-assisted technologies in the writing process

The generative-AI tools used in making this paper were Anthropic Claude (Opus 4 family and Sonnet 5, via the Anthropic API and Claude Code) and OpenAI ChatGPT (GPT-5 family, web and API). They were used in five scoped ways, with the authors reviewing, revising, and testing every retained output: (i) assisting with drafting, editing, and formatting the paper’s prose; (ii) drafting the prompt-injection corpus (Section 4.3.1) against a documented per-class hook list, every prompt hand-reviewed, following InjecAgent [40]; (iii) drafting boilerplate harness code, with all attack and aggregation logic verified by the authors; (iv) coding assistance for the authors’ implementation of the extension code (Claude Code); the

released open-source SDK was then derived from that code with AI assistance to strip internal references; and (v) surfacing candidate references, each of which the authors verified against its source (arXiv ID, DOI, ACL Anthology, IEEE Xplore, or publisher page) for title, authors, year, and venue—discarding any that could not be

cross-checked, with an automated re-verification pass before submission. These same models are also subjects of the evaluation in Section 4.3.1, as disclosed there. AI tools were *not* used to fabricate results, invent citations, or substitute for authorial reasoning; use complied with the ACM Policy on Authorship [6], and the authors take full responsibility for the paper.