Unified Plugin Architecture for Enterprise AI Agents: A Model Context Protocol Approach

Kaman Research

Version 1.0 · March 2026 · Production

Abstract

Enterprise AI agents must interact with an ever-expanding ecosystem of external systems—databases, SaaS platforms, internal APIs, messaging services—yet the dominant integration paradigms remain brittle, provider-specific, and resistant to composition. This paper presents a unified plugin architecture built on the Model Context Protocol (MCP) that addresses the *tool fragmentation problem*: the combinatorial explosion of bespoke integrations required when N agent capabilities must connect to M external systems.

We introduce a hub-and-spoke architecture comprising four layers: a **transport abstraction layer** supporting four wire protocols (STDIO, HTTP, SSE, WebSocket) with unified lifecycle management; a **function catalog** with vector-embedded semantic search enabling agents to discover relevant tools from thousands of registered capabilities; a **multi-scope credential vault** providing tenant-isolated credential injection with automatic token lifecycle management; and a **sandboxed execution runtime** that enforces strict security boundaries while maintaining the flexibility required for arbitrary plugin logic.

The architecture reduces the integration surface from $O(N \times M)$ bespoke connectors to $O(N + M)$ protocol-conformant adapters, while providing enterprise-grade security guarantees including multi-tenant isolation, role-based access control, and cryptographic credential management. We report on production deployment experience supporting 39+ MCP connectors across database, SaaS, and communication domains, and discuss the design tradeoffs that distinguish this approach from prior work in tool-augmented language models.

1 Introduction

1.1 The Tool Fragmentation Problem

The promise of AI agents—autonomous software that can reason, plan, and act on behalf of users—depends critically on their ability to interact with external systems. A procurement agent must query databases, send approval emails, update ERP records, and generate reports. A customer support agent must search knowledge bases, create tickets, escalate to humans, and log interactions. Each of these capabilities requires integration with a distinct external system, each with its own authentication model, data format, error semantics, and transport protocol.

Prior work has addressed this challenge through three approaches, each with fundamental limitations:

Hardcoded integrations. Early agent frameworks embedded API calls directly in agent logic. This creates an $O(N \times M)$ integration matrix: each of N agent capabilities requires a custom adapter for each of M target systems. Adding a new database vendor requires modifying every agent that performs data operations. The maintenance burden grows quadratically.

REST wrapper libraries. Tool libraries such as those in LangChain and Semantic Kernel provide pre-built wrappers around popular APIs. While reducing initial development cost, these libraries create tight coupling between the agent framework and specific API versions. Version drift, authentication changes, and schema evolution require coordinated updates across the entire wrapper library. Furthermore, the wrapper abstraction often leaks: error handling, pagination, and rate limiting behaviors vary per provider and cannot be fully abstracted.

Function calling specifications. OpenAI's function calling and similar specifications define a schema for tool invocation but leave transport, discovery, credential management, and execution sandboxing entirely to the implementer. They solve the *invocation interface* problem but not the *integration architecture* problem.

1.2 Contributions

This paper presents a production-grade plugin architecture that reduces the integration surface to $O(N + M)$ by introducing a protocol-mediated abstraction layer between agents and external systems. Our contributions are:

1. A **transport-agnostic protocol layer** that unifies four wire protocols under a single lifecycle model, enabling plugins to be implemented in any language and deployed in any topology.
2. A **semantic function catalog** that uses vector embeddings for tool discovery, allowing agents to find relevant capabilities from large registries without exhaustive enumeration.
3. A **multi-scope credential architecture** that isolates credentials by tenant, team, and user while supporting automatic OAuth token refresh and template-based injection.
4. A **sandboxed execution model** that provides plugins with controlled access to HTTP clients, database connections, and LLM services while preventing unauthorized system access.
5. Production validation across 39+ connectors spanning databases, SaaS platforms, communication services, and data processing systems.

2 Design Principles

The architecture is governed by five principles derived from production experience with enterprise AI deployments:

Protocol-first design. The boundary between the agent and a plugin is defined by a message protocol, not by a programming language interface or a shared library. This enables polyglot plugin development: a Python data science plugin and a Go database connector conform to the same protocol and are indistinguishable to the agent runtime. Protocol-first design also enables independent versioning—plugins can evolve without requiring agent-side changes, provided they maintain protocol compatibility.

Transport agnosticism. Enterprise environments exhibit heterogeneous deployment topologies. Some plugins run as sidecar processes on the same host (favoring STDIO transport for minimal latency). Others run as shared microservices behind load balancers (favoring HTTP). Real-time data feeds require persistent connections (favoring SSE or WebSocket). The architecture must support all four without requiring plugin authors to implement transport negotiation logic.

Credential isolation. In multi-tenant environments, a single plugin (e.g., a PostgreSQL connector) may serve hundreds of tenants, each with distinct credentials. Credentials must never be visible to plugin code at rest. They must be injected at execution time, scoped to the requesting user's permission level, and automatically refreshed when OAuth tokens expire. The credential lifecycle must be managed independently of the plugin lifecycle.

Sandbox execution. Plugin code executes arbitrary logic—HTTP requests, data transformations, even code generation. The execution environment must provide sufficient capability for legitimate operations while preventing unauthorized file system access, network exfiltration, or resource exhaustion. The sandbox boundary must be enforced at the runtime level, not through code review or trust assumptions.

Eventual consistency over strong coupling. Plugin registrations, function catalogs, and credential bindings are distributed across multiple services. Rather than requiring synchronous coordination (which creates fragile distributed transactions), the architecture uses event-driven synchronization with idempotent handlers. A plugin registered in the marketplace eventually appears in the agent's tool catalog; a credential update eventually propagates to all active sessions. This trade-off accepts seconds of staleness in exchange for operational resilience.

3 Architecture Overview

3.1 Hub-and-Spoke Model

The architecture follows a hub-and-spoke topology where the *agent runtime* serves as the hub, and *MCP-conformant plugins* serve as spokes. Four subsystems mediate the interaction:

```
            Plugin Registry
                |
[MCP Plugin A] --+ | +-- [MCP Plugin C]
             | v |
[MCP Plugin B] --+-> Agent Runtime <--+
             | ^ | |
[MCP Plugin D] --+ | +-- [MCP Plugin E]
                |
         +--------+--------+
         | | |
     Function Credential Execution
```

```
      Catalog Vault Sandbox
```

Plugin Registry. The authoritative catalog of installed plugins, their transport configurations, and their health status. Plugins self-describe via a manifest that declares capabilities, required credentials, and supported transport protocols.

Function Catalog. A searchable index of all functions exposed by registered plugins. Each function entry includes a natural-language description, typed input/output schemas, and a vector embedding for semantic discovery. The catalog is the bridge between the agent's intent ("I need to query a database") and the specific function that fulfills it.

Credential Vault. A multi-scope credential store that manages encrypted credential profiles. Credentials are never stored in the plugin registry or function catalog—they are resolved at execution time based on the requesting user's identity and scope.

Execution Sandbox. The runtime environment in which plugin functions execute. It provides controlled access to a curated set of capabilities (HTTP clients, utility libraries, cryptographic functions) while enforcing resource limits and preventing unauthorized operations.

3.2 Registration-to-Execution Flow

The lifecycle of a plugin from registration to invocation follows a five-phase pipeline:

```
# Phase 1: Registration
Register plugin manifest (name, transport,
  capabilities, credential requirements)
  -> Store in Plugin Registry
  -> Emit registration event

# Phase 2: Function Discovery
For Each capability in manifest:
  Register function (schema, description)
  -> Generate vector embedding
  -> Store in Function Catalog

# Phase 3: Connector Binding
Create connector instance
  -> Bind credential profile
  -> Configure transport endpoint
  -> Validate connectivity

# Phase 4: Tool Resolution (at agent runtime)
Agent receives user query
  -> Semantic search over Function Catalog
  -> Resolve matching functions
  -> Bind as structured tools with schemas

# Phase 5: Execution
Agent invokes tool
  -> Resolve credentials for user scope
  -> Inject into execution context
  -> Execute in sandbox with timeout
  -> Return structured result to agent
```

Each phase is decoupled from the others. A plugin can be registered without any connector bindings (Phase 1–2 only). Connectors can be created before or after function discovery. Tool resolution happens at agent runtime, not at registration time, enabling dynamic capability discovery.

4 Transport Protocol Abstraction

4.1 Supported Transports

The MCP specification defines a message-level protocol independent of the underlying transport. Our implementation supports four transports, each optimized for different deployment topologies:

Transport	Use Case	Latency	Topology
STDIO	Co-located plugins, CLI tools	<1 ms	1:1
HTTP	Stateless microservices	5–50 ms	N:1
SSE	Streaming data, long-running ops	5–50 ms	N:1
WebSocket	Bidirectional, real-time	2–10 ms	N:1

Table 1: Transport characteristics by deployment topology.

4.2 Transport Selection Criteria

Transport selection is not arbitrary—it reflects fundamental tradeoffs in the deployment model:

STDIO provides the lowest latency and simplest deployment model (the plugin runs as a child process), but creates a 1:1 coupling between the agent instance and the plugin process. This is appropriate for plugins that are lightweight, stateless, and tightly coupled to a single agent (e.g., a local file system browser or a code formatter).

HTTP enables shared plugin services behind load balancers, supporting N:1 fan-in from multiple agent instances. The stateless request-response model simplifies horizontal scaling but introduces network latency and precludes server-initiated messages. Most production database connectors and SaaS integrations use HTTP transport.

SSE (Server-Sent Events) extends HTTP with a persistent unidirectional channel from plugin to agent. This is essential for plugins that produce streaming results—large query result sets, real-time data feeds, or long-running operations that report incremental progress. The client initiates via HTTP; the server responds with an event stream.

WebSocket provides full-duplex communication, enabling both agent-initiated requests and plugin-initiated notifications. This is used for plugins that require bidirectional interaction, such as collaborative editing tools or interactive debugging sessions. The persistent connection introduces state management complexity but enables the richest interaction patterns.

4.3 Connection Lifecycle Management

Regardless of transport, every plugin connection follows a unified lifecycle:

1. **Initialization**: Transport-specific handshake (process spawn for STDIO, HTTP upgrade for WebSocket, GET request for SSE, or simple POST for HTTP).
2. **Capability negotiation**: The agent queries the plugin's manifest to discover supported functions, resource types, and protocol version.
3. **Health monitoring**: Periodic heartbeat probes verify plugin liveness. Configurable timeouts trigger automatic reconnection with exponential backoff.
4. **Graceful shutdown**: The agent sends a termination signal; the plugin completes in-flight operations before closing the transport.

Connection parameters—timeout durations, retry policies, read timeouts for SSE streams—are configurable per plugin instance, allowing operators to tune for their specific latency and reliability requirements.

4.4 Failure Modes and Recovery

Production experience has revealed several transport-specific failure modes that the architecture explicitly handles:

Session termination during SSE streams. Long-running SSE connections may be interrupted by network proxies, load balancer idle timeouts, or plugin restarts. The architecture implements automatic reconnection with configurable read timeouts (default: 600 s) and request timeouts (default: 60 s). Failed operations are retried with exponential backoff up to a configurable maximum.

Endpoint migration. When plugins are redeployed (e.g., during rolling updates), their endpoint URLs may change. The plugin registry maintains a health-checked endpoint map and routes requests to healthy instances. Stale endpoint references are flushed on service restart.

Protocol version mismatch. Plugins and agents may run different protocol versions during rolling deployments. The capability negotiation phase includes version exchange; incompatible versions trigger a graceful degradation path rather than a hard failure.

5 Function Discovery and Semantic Search

5.1 The Discovery Problem

As plugin ecosystems grow, the number of available functions can reach into the hundreds or thousands. An agent processing a user request such as "check if the quarterly revenue report is ready" must identify which of potentially thousands of functions are relevant—without requiring the user to specify the exact function name or the developer to hardcode the mapping.

Traditional approaches use keyword matching or hierarchical taxonomies. Keyword matching fails on semantic equivalence ("fetch" vs. "retrieve" vs. "get"). Taxonomies require manual curation and become stale as new functions are added. Neither scales to large, heterogeneous function catalogs.

5.2 Vector-Embedded Function Catalog

Our approach embeds every function description into a dense vector space using a pre-trained embedding model. At query time, the agent's intent is embedded in the same space, and the nearest functions are retrieved via approximate nearest neighbor search.

The embedding process operates as follows:

1. When a function is registered, its natural-language description is extracted from the manifest.
2. The description is passed through an embedding model to produce a fixed-dimensional vector representation.
3. The vector is stored alongside the function metadata in a vector-capable database (PostgreSQL with the pgvector extension in our implementation).

4. At query time, the agent's inferred intent is embedded using the same model, and a cosine similarity search retrieves the top-k matching functions.

This approach handles synonyms, paraphrases, and conceptual similarity without manual synonym dictionaries or taxonomy maintenance.

5.3 Schema Normalization

Functions in the catalog expose a normalized schema that distinguishes three categories of parameters:

Dynamic parameters are values that the agent provides at invocation time based on the current context. These are the parameters that appear in the tool's schema and are visible to the language model during planning. Examples include a search query, a record identifier, or a date range.

Configuration parameters are values that a user provides once during setup—API keys, database connection strings, service endpoints. These are stored in the credential vault and injected at execution time. They are *never* visible to the language model and *never* included in the tool schema.

Output declarations document the structure of the function's return value. While not used for invocation, they enable the agent to reason about data flow when composing multi-step workflows.

This three-way separation is critical for security. If configuration parameters (which often contain credentials) were included in the tool schema, the language model might hallucinate or leak credential values. By architecturally excluding them from the schema, credential exposure is prevented at the protocol level rather than relying on prompt engineering.

5.4 Description Quality and LLM Comprehension

The function description serves a dual purpose: it is both the search target for semantic discovery *and* the basis for the language model's decision about when to invoke the tool. Production experience has shown that description quality is the single largest determinant of tool selection accuracy.

Effective descriptions follow a pattern: they state what the function does, what conditions warrant its use, and what the return value represents. Descriptions are length-bounded to prevent context window pollution when the agent binds multiple tools simultaneously. The system enforces a maximum description length, truncating with a semantic-preserving strategy that retains the opening sentence (which typically contains the most discriminative information).

6 Credential Management and Security

6.1 Multi-Scope Credential Profiles

Enterprise environments require credential granularity beyond simple per-user key stores. A database connector might use organization-wide read-only credentials for reporting but user-specific credentials for write operations. A SaaS integration might use team-level API keys with per-user OAuth tokens for impersonation.

The credential vault supports three scope levels with a defined resolution priority:

Scope	Semantics	Priority
User	Personal credentials, per-user OAuth	Highest
Team	Shared team API keys, group tokens	Medium
Organization	Org-wide service accounts	Lowest

Table 2: Credential scope resolution order.

When a function requires credentials, the vault resolves them by searching from the narrowest scope (user) to the broadest (organization), returning the first match. This enables a natural override pattern: an organization provides default credentials, teams override for their specific environments, and individual users override for personal accounts.

6.2 Credential Field Types

The credential system supports four authentication paradigms through typed credential fields:

OAuth 2.0 fields manage the complete OAuth lifecycle: authorization code exchange, token storage, automatic refresh before expiration, and scope management. The architecture monitors token expiration timestamps and proactively refreshes tokens before they expire, preventing mid-execution authentication failures.

OpenID Connect (OIDC) fields extend OAuth with identity verification, supporting enterprise

SSO flows where the plugin must authenticate as a specific user.

API Key fields store static key-value pairs with encryption at rest. These are the simplest credential type, used for services with key-based authentication.

Basic Authentication fields store username-password pairs, encrypted at rest and injected as HTTP Basic Auth headers or connection string parameters at execution time.

All credential field values are encrypted at rest using envelope encryption. The encryption key hierarchy ensures that a database breach does not expose plaintext credentials. Decryption occurs only at execution time, within the sandbox boundary, and decrypted values are never logged or persisted in execution traces.

6.3 Template-Based Credential Injection

Rather than passing raw credential objects to plugin code, the system uses template-based injection. Plugin manifests declare *which* credential fields they require and *how* they should be injected (as HTTP headers, connection string parameters, environment variables, or context object properties). At execution time, the sandbox resolves the credential profile, extracts the required fields, and injects them into the execution context according to the template.

This indirection provides two benefits: plugin code never contains credential-handling logic (reducing the attack surface), and credential rotation is transparent to plugins (the same template resolves to new values after rotation).

6.4 Sandbox Execution Model

Plugin functions execute within a controlled sandbox that provides a curated set of capabilities:

Capability	Description
HTTP Client	Outbound HTTP requests (configurable allowlist)
Utility Libraries	Data transformation, string manipulation
Cryptographic Primitives	Hashing, HMAC, signature verification
LLM Client	Proxied access to language models
Database Client	Scoped query execution via connectors

Table 3: Sandbox-provided capabilities.

The sandbox enforces several constraints: *execution timeout* prevents runaway computations; *memory limits* prevent resource exhaustion; *network restrictions* limit outbound connections to declared endpoints; and *module restrictions* prevent access to file system operations, process spawning, and other system-level APIs. The specific boundaries are configurable per deployment but default to a conservative posture.

The execution model follows a validate-contextualize-execute-return pipeline:

1. **Validate**: Input parameters are checked against the function's declared schema. Type mismatches, missing required fields, and constraint violations are rejected before execution begins.
2. **Contextualize**: The sandbox constructs an execution context containing injected credentials, configured clients, and utility references.
3. **Execute**: The function code runs within the sandbox with enforced timeouts. Uncaught exceptions are captured and wrapped in structured error responses.
4. **Return**: The function's return value is serialized and returned to the agent runtime. Credential values are scrubbed from any error messages or stack traces.

7 Event-Driven Synchronization

7.1 The Synchronization Challenge

The plugin architecture spans multiple services: a marketplace where plugins are authored and published, an agent runtime where they are executed, and a credential service where authentication is managed. These services maintain independent data stores optimized for their respective access patterns. Keeping them consistent requires a synchronization mechanism that is resilient to partial failures, idempotent under retry, and does not introduce tight coupling between services.

7.2 Event Streaming Architecture

We adopt an event-driven architecture using a distributed messaging system. Plugin lifecycle events—registration, update, deletion, capability changes—are published as structured events to named subjects. Downstream consumers process these events asynchronously with at-least-once delivery guarantees.

```
# Plugin saved in marketplace
Emit "extension_saved" {
  pluginId, name, type, manifest,
  functions[], transport, credentials
}

# Agent runtime consumer
  -> Upsert plugin record
  -> For Each function:
    -> Upsert function record
    -> Recompute vector embedding
    -> Update function catalog index

# Credential service consumer
  -> Register credential template
  -> Validate required fields
```

This decoupling means that the marketplace does not need to know about the agent runtime's internal data model, and the agent runtime does not need to call the marketplace API to discover new plugins. Both react to events independently.

7.3 Idempotency and Ordering

Event consumers are designed to be idempotent: processing the same event twice produces the same result as processing it once. This is achieved through content-addressed deduplication—each event carries a unique identifier derived from its content, and consumers check for prior processing before applying changes.

Ordering guarantees are provided per-plugin: events for the same plugin are delivered in order, while events for different plugins may be processed concurrently. This enables high throughput for bulk plugin registrations while ensuring that a rapid sequence of updates to a single plugin (e.g., register, update manifest, add functions) is applied in the correct order.

7.4 Webhook Subscriptions and Resource Monitoring

Beyond plugin lifecycle events, the architecture supports webhook-based subscriptions for monitoring external resources. Connector instances can register webhook endpoints that receive notifications when the underlying resource changes—for example, when a database schema is modified, when a new file appears in a monitored directory, or when a SaaS resource is updated.

Webhook subscriptions follow a subscription-notification-acknowledgment pattern:

1. The connector registers a webhook URL with the external system (or with an intermediary that polls the system).
2. When a change is detected, a notification is delivered to the agent runtime.
3. The agent runtime processes the notification, updates its internal state (e.g., refreshing cached schema information), and acknowledges receipt.

Stale webhook registrations—those pointing to endpoints that no longer exist—are detected through periodic health checks and flushed automatically on service restart.

7.5 Schema Change Detection

A particularly important class of resource change is schema evolution in connected data sources. When a database table gains a new column, or an API endpoint changes its response format, downstream workflows that depend on the previous schema may break silently.

The architecture addresses this through schema snapshots and change detection:

1. At sync time, the system captures a fingerprint of the source schema (column names, types, constraints).
2. On subsequent syncs, the current schema is compared against the stored fingerprint.
3. If changes are detected, the system classifies them (additive, breaking, or compatible) and notifies affected workflows.
4. Column mapping configurations are validated against the new schema, and incompatible mappings are flagged for manual review.

This proactive approach transforms schema drift from a runtime failure into a managed change event.

8 Data Lake Integration

8.1 MCP Resources as Data Sources

Beyond tool invocation, the MCP protocol supports *resource* semantics: plugins can expose structured data sets that the platform can ingest, cache, and query. This transforms the plugin architecture from a pure tool-calling interface into a data integration layer.

When a user registers a data source through an MCP plugin, the platform:

1. Connects to the external system via the plugin's transport layer.
2. Validates connectivity and credential sufficiency.

3. Discovers the available resources (tables, collections, endpoints).
4. Registers selected resources with the platform's data lake for caching and versioned access.

8.2 Resource Filtering and Selective Sync

Enterprise data sources often expose hundreds of resources (tables, views, API endpoints), but a given use case requires only a subset. Synchronizing everything would waste storage, bandwidth, and processing time. The architecture supports explicit resource filtering: users specify which URIs to sync, and the platform ignores all others.

This filtering is specified declaratively at data source creation time. The sync engine evaluates each discovered resource against the filter and skips non-matching resources. This approach avoids the common pitfall of "sync everything and filter later," which creates unnecessary load on both the source system and the data lake.

8.3 Incremental Synchronization

For large data sources, full synchronization on every cycle is prohibitively expensive. The architecture supports incremental sync modes:

Timestamp-based: The system tracks the maximum value of a designated timestamp column and fetches only rows modified after that watermark.

Change-data-capture: For sources that support CDC (e.g., PostgreSQL logical replication, database change streams), the platform consumes change events and applies them incrementally.

Hash-based: For sources without native change tracking, the system computes content hashes and detects changes through hash comparison.

The sync mode is configurable per data source, and the system maintains sufficient metadata to resume incremental sync after failures without data loss or duplication.

8.4 Schema Mapping and Transformation

Source schemas rarely match the target schema required by downstream consumers. The architecture provides a declarative column mapping layer:

```
# Source: external database table
Columns: [customer_id, full_name,
  created_at, status_code]

# Mapping declaration
customer_id -> id (rename)
full_name -> name (rename)
created_at -> created (rename + cast)
status_code -> status (rename + transform)

# Target: data lake table
Columns: [id, name, created, status]
```

Mappings are validated against both the source schema snapshot and the target schema. When schema changes are detected (Section 7.4), the system re-validates all affected mappings and flags those that have become incompatible.

9 Security Model

9.1 Multi-Tenant Isolation

The architecture enforces strict multi-tenant isolation at every layer. All data access—plugin registrations, function catalogs, credential profiles, data source configurations, execution logs—is scoped by a tenant identifier that is extracted from the authenticated session and cannot be overridden by client-supplied parameters.

Tenant isolation is enforced through a combination of application-level filtering (every query includes a tenant predicate) and database-level policies. Cross-tenant data access is architecturally impossible: there is no API that accepts a tenant identifier as a parameter.

9.2 Role-Based Access Control

The platform's RBAC model extends to plugin operations:

Operation	Required Permission
Register plugin	Administrator or Plugin Developer
Create connector	Administrator or Integration Manager
Bind credentials	Credential owner or Administrator
Invoke function	Any authenticated user with tool access
View execution logs	Auditor or Administrator
Manage data sources	Data Steward or Administrator

Table 4: RBAC permission model for plugin operations.

Permissions are evaluated at the API gateway level before requests reach the plugin subsystem. This ensures that authorization is enforced consistently regardless of the entry point (UI, API, agent runtime, or event consumer).

9.3 Execution Audit Trail

Every plugin invocation generates an audit record containing: the invoking user's identity, the function invoked, a hash of the input parameters (not the parameters themselves, to avoid logging sensitive data), the execution duration, the outcome (success or failure class), and a correlation identifier linking the invocation to the agent session that triggered it.

Audit records are append-only. The application's database role for the audit table permits `INSERT` and `SELECT` operations only—`UPDATE` and `DELETE` are revoked. This provides tamper-evidence: once an audit record is written, it cannot be modified or removed through the application layer.

9.4 Credential Security Properties

The credential management system provides several formal security properties:

- **Encryption at rest**: All credential field values are encrypted using envelope encryption with a key hierarchy. Database-level access does not expose plaintext credentials.
- **No credential logging**: Credentials are scrubbed from all log output, error messages, and exception stack traces before they leave the sandbox boundary.
- **Scope-limited visibility**: A user can only resolve credentials at their scope level or broader. User-scoped credentials are invisible to other users; team-scoped credentials are invisible to other teams.
- **Temporal validity**: OAuth tokens carry expiration metadata. The system refuses to use expired tokens and initiates refresh automatically, preventing the accumulation of stale credentials.

10 Performance and Scalability

10.1 Latency Characteristics

Plugin invocation latency comprises four components: credential resolution, transport overhead, plugin execution, and result serialization. The following table summarizes observed latencies in production:

Component	P50	P95	P99
Credential resolution	2 ms	8 ms	15 ms
Transport (STDIO)	<1 ms	1 ms	3 ms
Transport (HTTP)	5 ms	25 ms	50 ms
Transport (SSE setup)	10 ms	40 ms	80 ms
Sandbox initialization	3 ms	10 ms	20 ms
Semantic search (top-10)	8 ms	25 ms	50 ms
Schema validation	1 ms	3 ms	5 ms

Table 5: Invocation latency breakdown (platform overhead only; excludes plugin execution time).

The dominant latency contributor is the plugin's own execution time, which varies from single-digit milliseconds for simple transformations to seconds for complex database queries or external API calls. The platform overhead (credential resolution + transport + sandbox + validation) consistently remains below 100 ms at P99, ensuring that the architecture does not become the bottleneck.

10.2 Connection Pooling

For HTTP and WebSocket transports, the architecture maintains connection pools to avoid the overhead of repeated TLS handshakes and TCP connection establishment. Pool parameters—maximum connections per plugin, idle timeout, maximum connection lifetime—are configurable per plugin instance.

Connection pools are partitioned by tenant to prevent resource contention: a high-traffic tenant's connection usage cannot starve other tenants. Pool exhaustion triggers graceful degradation (queuing with bounded wait time) rather than hard failure.

10.3 Horizontal Scaling

The architecture is designed for stateless horizontal scaling at every layer:

Agent runtime: Multiple instances behind a load balancer. No affinity required—any instance can resolve any plugin invocation because the function catalog, credential vault, and plugin registry are shared services.

Plugin services: HTTP and SSE plugins scale independently via standard container orchestration. The plugin registry maintains a health-checked endpoint list and distributes requests across healthy instances.

Event consumers: Event processing scales via consumer groups. Adding a consumer instance increases throughput linearly without requiring configuration changes.

Semantic search: Vector similarity search scales horizontally through database read replicas and index partitioning. The embedding computation is stateless and can be distributed across GPU-equipped workers for high-throughput bulk indexing.

10.4 Catalog Scaling Properties

The function catalog's performance characteristics are critical because semantic search occurs on every agent interaction. We observe the following scaling behavior:

$$T_{\text{search}} \approx c_1 \cdot \log(n) + c_2 \cdot k \tag{1}$$

where n is the number of functions in the catalog, k is the number of results requested, c_1 is the index traversal cost (dependent on the HNSW index parameters), and c_2 is the result materialization cost. The logarithmic dependence on catalog size means that search latency grows slowly even as the function catalog scales to tens of thousands of entries.

11 Related Work

11.1 Tool-Augmented Language Models

The concept of augmenting language models with external tools has been explored extensively in recent literature. Toolformer demonstrated that language models can learn to use tools through self-supervised training. ReAct introduced the reason-and-act paradigm where models alternate between reasoning traces and tool invocations. These works establish the *cognitive* foundation for tool use but do not address the *systems* challenges of managing large tool ecosystems in production.

11.2 LangChain and LangGraph

LangChain provides a tool abstraction layer with a large library of pre-built integrations. Tools are defined as Python classes with typed input schemas and are bound to agents at initialization time. While comprehensive, this approach has several limitations that our architecture addresses:

- **Language coupling**: LangChain tools must be implemented in Python (or JavaScript in LangChain.js). Our protocol-first approach enables plugins in any language.
- **Static binding**: Tools are bound at agent initialization, not discovered dynamically. Our semantic search enables runtime tool discovery from large catalogs.
- **Credential management**: LangChain defers credential management to the developer. Our multi-scope vault provides enterprise-grade credential lifecycle management.
- **Execution isolation**: LangChain tools execute in the same process as the agent. Our sandbox provides resource limits, timeout enforcement, and capability restrictions.

LangGraph extends LangChain with stateful, graph-based agent workflows. Our architecture integrates with LangGraph at the tool binding layer: MCP functions are wrapped as LangGraph-compatible structured tools, enabling the workflow engine to invoke plugin functions as graph nodes.

11.3 OpenAI Function Calling

OpenAI's function calling specification defines a JSON Schema-based interface for declaring tool capabilities and parsing model-generated invocations. This specification solves the *invocation interface* problem elegantly but leaves several enterprise requirements unaddressed:

- **No transport protocol**: Function calling defines what to call, not how to reach it.
- **No credential management**: Authentication is entirely the developer's responsibility.
- **No discovery mechanism**: Available functions must be enumerated explicitly; there is no semantic search.
- **No execution model**: The specification assumes the developer implements execution; there is no sandbox or resource management.

Our architecture is compatible with OpenAI's function calling format at the schema level—MCP function schemas can be serialized as OpenAI function definitions—but provides the surrounding infrastructure that production deployments require.

11.4 Semantic Kernel

Microsoft's Semantic Kernel provides a plugin model where "skills" are organized into named groups with typed parameters. Semantic Kernel's approach is closest to ours in its recognition that tool management is a first-class concern, but differs in several ways: it assumes a .NET or Python runtime, uses explicit skill registration rather than semantic discovery, and does not provide multi-tenant credential isolation.

11.5 Model Context Protocol

The Model Context Protocol, introduced by Anthropic, defines a standardized protocol for communication between AI applications and external tools. Our architecture adopts MCP as the wire protocol while extending it with the enterprise infrastructure layers (credential management, semantic discovery, execution sandboxing, event-driven synchronization) that production deployments require. We view MCP as the *necessary* foundation and our architecture as the *sufficient* enterprise wrapper.

11.6 Comparative Summary

Capability	*Kaman MCP*	*LangChain*	*OpenAI FC*	*Sem. Kernel*
Multi-transport	✓	—	—	—
Semantic discovery	✓	—	—	—
Multi-scope credentials	✓	—	—	—
Execution sandbox	✓	—	—	Partial
Multi-tenant isolation	✓	—	—	—
Polyglot plugins	✓	—	—	Partial
Event-driven sync	✓	—	—	—
Data lake integration	✓	—	—	—
Schema change detection	✓	—	—	—
RBAC integration	✓	—	—	Partial

Table 6: Capability comparison across tool integration approaches.

12 Conclusion and Future Work

12.1 Summary

We have presented a unified plugin architecture for enterprise AI agents that addresses the tool fragmentation problem through protocol-mediated abstraction. The architecture reduces the integration complexity from $O(N \times M)$ bespoke connectors to $O(N + M)$ protocol-conformant adapters by introducing four infrastructure layers: transport abstraction, semantic function discovery, multi-scope credential management, and sandboxed execution.

The key insight is that tool integration for enterprise AI agents is not primarily a *programming* problem—it is an *architecture* problem. Individual integrations are straightforward; the challenge lies in managing hundreds of integrations across multiple tenants with diverse credential requirements, evolving schemas, and strict security constraints. By treating integration management as a first-class platform concern rather than an application-level responsibility, the architecture enables sustainable growth of the plugin ecosystem without proportional growth in maintenance burden.

Production experience with 39+ MCP connectors validates the approach: new plugins can be developed, registered, and made available to agents without modifying the agent runtime, the credential infrastructure, or the discovery mechanism. The protocol-first design has proven particularly valuable for third-party plugin development, where plugin authors need not understand the agent's internal architecture—only the MCP protocol specification.

12.2 Future Directions

Several avenues for future work emerge from the current architecture:

Compositional tool chains. Current tool invocation is atomic: the agent calls one function at a time and orchestrates sequences through its planning loop. Future work will explore declarative tool composition, where multi-step operations (e.g., "query database, transform results, send email") can be expressed as a single composite tool with transactional semantics.

Federated plugin discovery. The current function catalog is centralized within a single platform deployment. Federated discovery would enable cross-organization plugin sharing: an organization could publish selected plugins to a shared catalog, enabling other organizations to discover and invoke them (subject to credential and access controls).

Adaptive tool selection. Semantic search provides a static relevance ranking. Future work will incorporate execution history—success rates, latency distributions, error patterns—into the ranking function, enabling the system to prefer tools that have historically performed well for similar queries.

Plugin observability and cost attribution. As plugin ecosystems grow, understanding which plugins consume the most resources, which generate the most errors, and which provide the most value to users becomes critical for operational management. Future work will integrate plugin-level observability with the platform's cost attribution system.

Formal verification of sandbox boundaries. The current sandbox relies on runtime enforcement of capability restrictions. Future work will explore formal verification techniques to provide stronger guarantees about sandbox isolation, particularly for plugins that execute user-supplied code.

Protocol evolution. As the MCP specification evolves, the architecture must accommodate new protocol features (e.g., streaming tool results, tool-initiated agent callbacks, resource subscriptions) while maintaining backward compatibility with existing plugins. A versioned protocol negotiation mechanism is under development to support graceful migration across protocol versions.