Scalable LLM Agent Tool Access in the Cloud

Mingxin Li[†‡*], Enge Song[‡*], Yueshang Zuo[‡*], Xiaodong Liu[‡], Rong Wen[§‡], Qiang Fu[¶], Gianni Antichi[°], Jian He[‡], Jing Tie[‡], Zhou Shao[‡], Xiaobo Xue[‡], Xiong Xiao[‡], Luyao Zhong[‡], Shaokai Zhang[‡], Jiangu Zhao[‡], Jianyuan Lu[‡], Shize Zhang[‡], Xiaoqing Sun[‡], Changgang Zheng[‡], Zihao Fan[‡], Haonan Li[‡], Tian Pan[‡], Xiaomin Wu[‡], Yang Song[‡], Xing Li[★‡], Biao Lyu[‡], Meng Li[†], Haipeng Dai[✉†], Guihai Chen[†], Shunmin Zhu[✉‡]

[†]Nanjing University [‡]Alibaba Cloud [§]Fudan University [¶]RMIT University
[°]Politecnico di Milano [★]Zhejiang University

Abstract

LLM agents increasingly rely on tool calling to act on external systems, and the Model Context Protocol (MCP) has quickly become its de facto interface. Operating MCP at cloud scale, however, becomes difficult. On the tool provider side, legacy services are not directly callable through MCP; the rapid protocol development also creates ongoing compatibility cost. On the agent side, the number of accessible tool is limited by the LLM context window and inference overhead; mounting a large tool set increases token usage and inference latency and can reduce task success rate. Moreover, for stateful MCP backends with multiple replicas, preserving session affinity increases client-side complexity.

We present a cloud-scale gateway system for MCP service. It breaks the direct-connect model on the data plane and offloads legacy service integration, consolidating incompatible MCP variants, access control, tool recommendation, and session-aware routing to the gateway. Hybrid retrieval sustains 98% Top-15 recall; it scales agent tool access to 3,000+ with high tool selection accuracy, and reduces tool selection time by 8.9× and token usage by 23.8×, with low per-call overhead, stable under scale-out. Finally, we share the lessons learned from deploying the gateway system in production.

1 Introduction

Large Language Model (LLM) applications are evolving from traditional chatbots to tool-using agents. Tool calling allows an LLM to access external systems and complete complex tasks, such as querying databases or managing cloud network resources [19, 30–32, 45, 56, 57, 60, 64]. Without a unified tool interface, each LLM agent had to be adapted to individual tools. This motivates standardizing the tool-invocation interface, resulting in the Model Context Protocol (MCP) [16]. MCP standardizes both how an agent invokes tools and how tool providers expose tools to agents (Figure 1). This lowers the engineering overhead for tool providers and enables rapid growth in the number of tools available to agents (reaching $O(10k)$ MCP servers within 13 months [35, 36]). Also, MCP traffic in the cloud is growing rapidly (Figure 2). Agents then rely on the LLM to interpret tool descriptions,

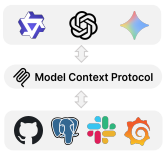


Figure 1. MCP

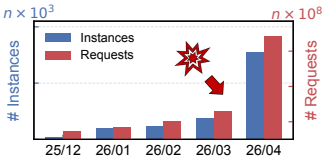


Figure 2. Traffic in one cloud region

select appropriate tools, and issue tool calls, enabling them to automate tasks that previously required manual operations.

However, building a super agent that can access massive tool sets with reliable tool invocation faces several challenges. (1) OpenAPI is the dominant interface in the web era. After more than a decade of adoption, the number of API servers has reached 10^6 [51], yet these existing APIs are not directly callable by agents through MCP. (2) Tool developers must handle work orthogonal to tool functionality, such as compatibility issues from rapid MCP evolution and the integration of authentication. (3) Agents depend on the LLM to understand tool descriptions and perform tool selection and invocation. Due to the limited context window, a single agent can only mount a limited number of tools, insufficient for diverse tasks at scale. Moreover, mounting too many tools increases token cost and processing latency while reducing task success rate. (4) MCP assumes a client–server direct-connect model. For availability and load balancing, MCP servers are often deployed with multiple replicas, which forces the Host to manage request distribution across multiple clients over replicas. For stateful servers, the Host must ensure *session-aware routing* (i.e., all requests in the same logical session are forwarded to the replica that maintains the session context).

To address these issues, we introduce a gateway system on the data plane, breaking the direct-connect model between client and server. While MCP runs over HTTP, its key routing information reside in the HTTP body rather than headers, preventing existing web-oriented application load balancers (e.g., L7 Load balancer and API Gateways) [7–10] from being directly reused without non-trivial customization. We implement the following capabilities: (1) *Protocol Adaptation.* We build a translation module that converts MCP tool calls into calls to existing API servers (enabling access to legacy APIs) and bridges incompatible MCP variants to reduce developer adaptation cost. (2) *Function Offloading.* To support fine-grained tool access control at scale,

[*]Mingxin Li, Enge Song, and Yueshang Zuo contributed equally.
[✉]Corresponding author: Haipeng Dai and Shunmin Zhu.

we implement authentication modules that enable precise identity-to-tool mappings. (3) *Tool Recommendation.* We design query-based tool pre-filtering for MCP. Beyond standard semantic embeddings, we exploit the fact that MCP tool names often explicitly encode functionality and propose a hybrid matching method combining semantic and lexical matching. Compared to LLM-based recommendation, this approach is deterministic, efficient, and scalable, making it suitable for deployment at the gateway without introducing long latency or unstable results. To address the drop in recommendation accuracy when generic embeddings fail to capture domain-specific knowledge in queries, we further design a query-rewrite method based on domain knowledge extraction. (4) *Session-aware Routing.* The gateway offloads request distribution from the Host to the gateway. To route requests to backend replicas that hold the corresponding session context, the gateway maintains a session-to-server mapping. Since cloud gateways are deployed with multiple instances, the gateway synchronizes the session-to-server mapping via a centralized store. Our key contributions are:

- **We present a deployed cloud-scale gateway system for massive MCP tool access.** Protocol adaptation enables direct access to legacy API servers with sub-millisecond conversion overhead (P50 $\leq 143\,\mu s$). The tool recommendation module achieves 98% recall for Top-15 tools when users mount $O(1k)$ tools, with latency below 250 ms. This improves agent tool selection accuracy, reduces time cost by 8.9× and token cost by 23.8×.
- **We address stateful request routing in multi-gateway deployments.** A centralized store-based session synchronization mechanism is proposed to support session-aware routing. We further introduce a Pub/Sub mechanism to synchronize responses across gateway nodes to preserve long-lived connections for stateful backends. Further, we show the overhead, and explore the trade-offs of session management strategies.
- **We share production deployment lessons.** (1) Maintaining long-lived gateway–backend connections reduces mean/P90/P99 latency by 51.7%/55.7%/56.5%, respectively. (2) Leveraging tool dependencies and tool-calling locality, knowledge caching returns complete prerequisite tool chains (100% completeness), and LRU semantic caching reduces recommendation latency on average by up to 60%. (3) We further discuss an encoding-based session synchronization scheme for efficient cross-gateway session coordination, and (4) propose an automated rule-extraction method for domain-aware query rewriting, improving tool recommendation accuracy from 80.8% to 99.7%.

2 Background and Motivation

2.1 Tool-Using LLM Agents

LLM-powered applications are evolving from conversational chatbots to general-purpose autonomous agents [24, 31, 34, 69]. Unlike early chatbots that focused on prompt-based text generation without external interaction, LLM agents can plan, use tools, and accomplish complex goals. Using reasoning-and-acting paradigms such as ReAct, CoT, and ToT, agents decompose user intents into multi-step tasks [61, 67, 68] and interact with external systems—for example, querying databases, managing cloud resources, and triggering workflows [19, 30, 31, 45, 56, 57, 60, 64]. The LLM acts as a controller that selects tools, emits structured arguments, and triggers execution (i.e., tool calls) [42, 50, 63]. As a result, LLM agents require robust tool infrastructure to mediate agent–tool interactions [13]. The absence of widely adopted protocols remains a major bottleneck to interoperating with external tools and scaling agent ecosystems, motivating recent standardization efforts [66]. The MCP [39] addresses this challenge by defining a standard interface between agents (*MCP Hosts*) and external resources (*MCP Servers*), facilitating tool integration.

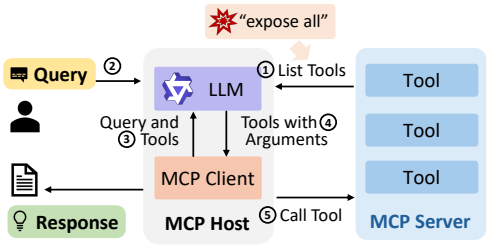


Figure 3. A concise MCP interaction workflow

2.2 The Rise of Model Context Protocol

Interoperability via interface standardization. Before the MCP, LLM applications and agent frameworks [15, 21, 22] typically integrated external tools through bespoke adapters, resulting in a fragmented ecosystem [27]. Heterogeneous tool interfaces and provider-specific calling schemas forced point-to-point integration, so connecting M model providers or frameworks to N tools conceptually approached an $O(M\times N)$ engineering burden. MCP reduces this complexity to $O(M+N)$ by standardizing the protocol boundary between LLM hosts and tool infrastructures: tools are exposed through a uniform, provider-agnostic interface, enabling LLM applications to access diverse tools without per-tool or per-provider rewrites [39]. MCP standardizes interoperability rather than meaning: it does not prescribe tool semantics, backend implementations, or the host's planning and orchestration logic.

MCP interaction workflow. An MCP host (e.g., an LLM application or agent) maintains a dedicated MCP client for each connected MCP server. As shown in Figure 3, the workflow starts with connection establishment and initialization. The client invokes *List Tools* to fetch tool definitions from one or more servers (①). Given a user query (②), the host combines the query with these definitions to form a composite prompt (③), then invokes the LLM to select tools and

produce structured arguments (④). The host parses the arguments and executes the selected tools via *Call Tool* (⑤). It repeats this select–execute loop until the LLM judges the request satisfied and returns the final response. Appendix §C provides a step-by-step description.

MCP protocol layers. MCP defines message semantics at two layers: (1) **Data Layer**, representing MCP primitives (e.g., *tools*, *resources*, *prompts*, and *notifications*) and lifecycle management as JSON-RPC messages; and (2) **Transport Layer**, specifying how these JSON-RPC messages are exchanged between clients and servers. The transport layer currently supports two options: HTTP+SSE and Streamable HTTP. *HTTP+SSE* is inherently stateful: the client establishes a long-lived SSE channel for server-to-client delivery, sends client-to-server JSON-RPC via HTTP POST, and includes a server-assigned Session-ID on each request; responses are streamed over the single, persistent SSE connection. *Streamable HTTP* uses a single MCP endpoint, and supports both stateless and stateful modes: in stateless mode, each POST is independent and no session is maintained; in stateful mode, the server assigns a Session-ID to maintain a logical session across requests without requiring a long-lived connection. These transport variants and session semantics directly shape the deployment challenges we characterize next (§2.3).

2.3 Challenges of Deploying MCP

Months of MCP service deployment experience indicate that deploying MCP at production scale is not a drop-in upgrade.

Legacy infrastructures. Large cloud platforms are primarily built around OpenAPI/Swagger endpoints, engineered for developers and conventional SDK ecosystems rather than native tool use by autonomous agents. This creates a structural mismatch in maturity and supply. OpenAPI-based infrastructure has been evolving for more than a decade since the first specification release [52], whereas MCP has existed for only about one year since its public release [6]. The MCP community registry currently contains on the order of 10^4 MCP servers [35, 36]. This is negligible compared with the OpenAPI ecosystem: as of January 2024, APIstic reports over 10^6 (1,275,568) valid OpenAPI/Swagger specifications mined from public sources [51]. Alibaba Cloud alone exposes on the order of 10^4–10^5 OpenAPI operations [3, 58, 59] across heterogeneous products (Figure A1), suggesting that the global OpenAPI surface is orders of magnitude larger than the current MCP. A practical solution must bridge legacy APIs into tool use in a cost-effective and non-intrusive manner.

> ***Challenge 1:*** *Large-scale legacy OpenAPI infrastructures are not MCP-native, making most production APIs inaccessible to agent tool use.*

Incompatible protocol versions. MCP is evolving rapidly, with four iterations released over a 13-month period, and MCP deployments already face incompatibilities along two fast-moving axes. First, transport-layer divergence. The ecosystem currently spans both the legacy HTTP+SSE transport and the newer Streamable HTTP transport (§2.2). Measurements from a commercial MCP hosting platform with 184 deployed servers [43] show that only a small fraction of servers support both transports, while the rest support only one. Client support further lags behind: in the public market, over 50% of MCP clients support only HTTP+SSE and lack effective support for Streamable HTTP [25]. Second, authentication divergence. Early MCP specifications did not define a standard authentication mechanism [16], leading servers to adopt heterogeneous, vendor-specific security methods. As authentication requirements evolved, deployments must absorb this heterogeneous authentication ecosystem to preserve compatibility, rather than requiring server developers to repeatedly re-implement authentication logic.

Table 1. Context window of frontier foundation models, along with the maximum number of tools that can be mounted in model context window[1].

Model	Context Window	Max #Tools
Gemini-3-Flash [23]	**1M**	774
Qwen-Plus [54]	**1M**	627
GPT-5.2 [44]	**400k**	270
Qwen3-MAX [55]	**256k**	198
Claude-Opus-4.5 [5]	**200k**	125
Deepseek-V3.2 [18]	**128k**	98

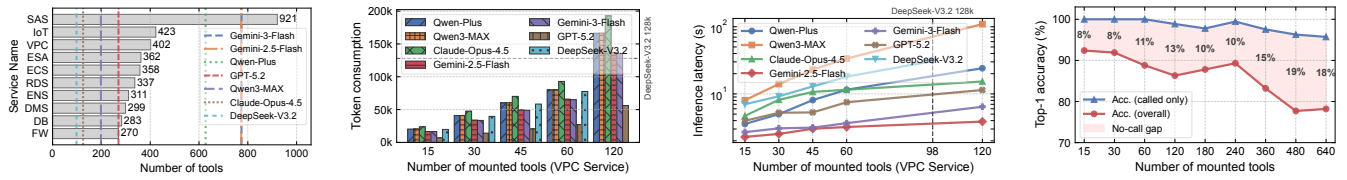
Table 2. For the VPC agent with 60 tools, a duplicated VPC task requires ∼10 tool-selection rounds. In round 1, the KV cache misses, while in rounds 2–10 it hits.

Model	Token Usage	Estimated Price
Claude-Opus-4.5 [4]	92,924 × 10	$0.999
Qwen3-MAX [55]	80,085 × 10	$0.685
GPT-5.2 [44]	27,100 × 10	$0.090
Gemini-3-Flash [23]	65,016 × 10	$0.061
DeepSeek-V3.2 [18]	77,564 × 10	$0.413
Qwen-Plus [54]	79,947 × 10	$0.090

> ***Challenge 2:*** *MCP evolution causes transport and authentication incompatibilities across versions, fragmenting deployments and increasing compatibility overhead.*

The expose-all tool approach. MCP hosts (e.g., LLM applications or agents) typically inline all tool schemas into the prompt (§2.2) and rely on the LLM to select tools and generate invocation arguments. Although state-of-the-art models support increasingly large context windows, context remains a constrained resource. Table 1 shows that even models with 1M-token context windows effectively saturate at roughly 600–800 tools. Because the context window must also accommodate dialogue history and intermediate reasoning, this "expose all" approach fails to scale to hyperscale tool sets for four reasons. **(1) Limited tool capacity.** Modern cloud platforms expose a broad range of services and thousands

[1]Measured in our experiments using a tool set derived from the Alibaba Cloud VPC, ECS and other services.



(a) **(b)** **(c)** **(d)**

Figure 4. The measured impact of tool scale. **(a)** # tools may exceed the capacity of models. **(b)** Inlining schemas incurs context overhead. **(c)** The scale of the tools leads to huge inference latency. **(d)** Tool-selection accuracy degrades.

of distinct operations. Even a single service category can include hundreds of tools (Figure 4a), making it difficult for models to accommodate tools from multiple categories. For example, an agent supporting only Security Center Service already requires over 921 tools, exceeding the practical capacity of all frontier models. **(2) Excessive schema token usage.** Tool schemas consume a disproportionate share of the context budget. Inlining just 120 VPC tools requires over 166k tokens for models such as Qwen-Plus and Qwen3-MAX, or more than 56k tokens even with efficient tokenization (Figure 4b). This verbosity increases inference cost (Table 2) and crowds out user queries, dialogue history, and intermediate reasoning. **(3) Prohibitive inference latency.** Prompt size grows with the tool set, slowing tool selection and argument generation. Figure 4c shows that for Qwen3-MAX, mounting 120 tools already pushes end-to-end latency (prompting + tool selection + argument generation) beyond 100 seconds, while 180 tools exceed 200 seconds. Such delays break interactivity and make real-time agent responses impractical. **(4) Degraded accuracy of tool use.** As tool schemas occupy an increasing fraction of the context, models become less reliable at selecting correct tools and generating valid arguments, as shown in Figure 4d. Specifically, the model increasingly selects inappropriate tools or produces malformed invocations, and, more notably, refuses to call any tool (no-call rate: 8.0%→19.0%). These failures are analogous to the "lost-in-the-middle" phenomenon [12, 33].

> ***Challenge 3:*** *The expose-all tool approach does not scale: large tool sets overwhelm the context window and incur prohibitive token cost, latency, and accuracy degradation.*

Distributing MCP requests across replicas is non-trivial. To achieve high availability and throughput with a massive number of tool calls, MCP services typically run multiple MCP server replicas and must distribute requests across them. In stateful deployments, MCP servers maintain session state across requests, so routing must ensure that all requests in the same session go to the replica holding the context (i.e., *session affinity*). This requirement makes conventional per-request load-balancing (e.g., round-robin) insufficient. As a result, hosts/clients often need to implement their own session-aware load-balancing and health-check/failover modules, adding development and operational overhead.

> ***Challenge 4:*** *Stateful MCP sessions make load balancing and replication difficult, requiring session affinity and failure-aware routing to preserve per-session context.*

2.4 Design Goals

Enabling legacy API access via MCP. To address Challenge 1, the system should expose large-scale API operations as standard MCP tools without backend refactoring.
MCP server adaptation overhead reduction. To address Challenge 2, the system must act as a compatibility layer across heterogeneous MCP transports and authentication schemes, without modifications to either clients or backends.
LLM context-efficient tool use. To address Challenge 3, the system should scale tool use to large catalogs without placing all tools in the LLM context. The design goal is bounded context overhead, stable latency and cost, and robust tool selection/argument generation accuracy as the tool set grows.
Request distribution with session affinity. To address Challenge 4, the system should distribute requests across MCP replicas while preserving session affinity, without requiring clients to implement session-aware logic.

3 System Design

The default MCP client–server direct-connect model was not designed to support large-scale operation, governance, and heterogeneous infrastructures on the cloud. This model becomes fragile as tool-calling traffic grows, tool infrastructures become elastic and multi-tenant, and operators require uniform policy enforcement.
The MCP needs a gateway system. In the web-service era, Layer-7 gateways became a foundational building block for reliable client–server communication by centralizing connectivity management, load balancing, and governance. A similar shared point is needed in the agent-service era, where agents interact with heterogeneous tool infrastructures (e.g., API servers and MCP servers). Motivated by these needs and informed by our experience in networked services and Layer-7 load balancing as a major cloud provider, we design and deploy a production gateway system. The gateway breaks the original direct-connect model by interposing a shared control point between MCP clients and tool servers.

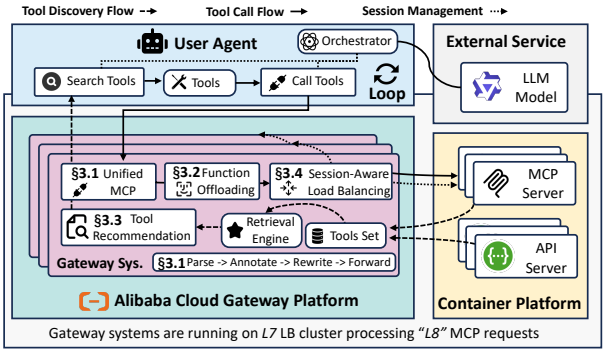


Figure 5. The gateway system architecture

System overview. The system (Figure 5) comprises four building blocks across the lifecycle of agent–MCP interaction. The gateway exposes heterogeneous backends, e.g., MCP transport variants and legacy APIs, through a unified MCP interface, so that a client connects to the gateway as a single MCP endpoint (§3.1), addressing Challenge 1 and the transport incompatibility in Challenge 2. On this unified path, the gateway authenticates the client and applies identity-based access control at ingress, and authenticates to backends on behalf of the client at egress, offloading these common functions from backends (§3.2), addressing the authentication incompatibility in Challenge 2. Instead of receiving thousands of tools, the agent discovers tools iteratively via the tool recommendation interface, which returns a small set of intent-relevant tools (§3.3), addressing Challenge 3. Subsequent tool-calling requests are routed to backends via session-aware load balancing, session state is synchronized across gateway replicas, so any gateway instance makes consistent forwarding decisions, preserving session context under replication (§3.4), addressing Challenge 4.

3.1 Protocol Adaptation for Heterogeneity

Tool backends are heterogeneous at both MCP layers (Challenges 1 and 2): at the data layer, legacy OpenAPI services do not implement MCP primitives; at the transport layer, backends diverge across conventional HTTP, HTTP+SSE, and Streamable HTTP. The gateway therefore provides a unified protocol abstraction across both layers. **(1) API protocol conversion.** To onboard legacy APIs as MCP tools at scale, the gateway performs adaptation at both layers. At the data layer, it translates MCP primitives (e.g., *List Tools* and *Call Tool*) into API invocations and normalizes responses to the MCP tool interface. Each OpenAPI operation is compiled into exactly one MCP tool. This one-to-one mapping minimizes tool granularity and reduces ambiguity in tool selection and argument generation. At the transport layer, it terminates the client-facing MCP connection and communicates with each backend using its required protocol. **(2) MCP transport bridging.** As MCP transport variants and versions are not strictly compatible, the gateway bridges between transport variants (e.g., HTTP+SSE and Streamable HTTP), enabling interoperability without changes to applications. **(3) Decoupled adaptation layers.** The gateway decouples transport bridging from data-layer mapping, rather than coupling them in a monolithic translator. Since MCP transports evolve rapidly (§2.3), this decoupling confines protocol churn to the transport stage: new transport variants can be supported without modifying data-layer conversion.

3.2 Function Offloading

In production deployments, existing tool backends often duplicate auxiliary logic (e.g., authentication and identity-based access control), which increases engineering cost and results in inconsistent enforcement across heterogeneous services. We therefore adopt gateway-side function offloading: the gateway provides these shared functions as common infrastructure and enforces them on the end-to-end request path. By ingress and egress authentication (Figure 5), the gateway offloads the following functions: **(1) Centralized security enforcement.** The gateway exposes a unified authentication interface on ingress and maps authenticated identities to backend-specific credentials on egress. For backends without native authentication, it enforces access policies at ingress and forwards requests over a protected gateway–backend path, establishing an end-to-end security boundary without backend changes. **(2) Fine-grained tool access control.** The gateway determines tool visibility and invocation permissions as a function of identity, enabling policy-driven tool exposure in multi-user MCP services. Overall, this design reduces backend complexity and ensures consistent enforcement across diverse tool infrastructures.

3.3 Intent-Driven Tool Recommendation

To avoid exposing the full tool catalog to the LLM context (Challenge 3), the gateway supports *intent-driven tool recommendation* (Figure 5): given the intent (e.g., task description or query context), it returns a small set of relevant tools from the massive backends. **(1) Dual-path indexing.** Cloud tools are described by both natural language and specialized identifiers (e.g., product and resource names); neither semantic retrieval nor lexical matching alone covers both. The gateway therefore adopts a semantic–lexical dual-path index, supporting fuzzy intent understanding while preserving precise keyword matching. **(2) Iterative discovery.** Tool discovery is exposed as an agent-invokable interface. The agent can iteratively search by its loop with or w/o LLM reasoning. **(3) Deterministic and efficient execution.** Instead of relying on LLM, tool recommendation runs gateway-side as deterministic, lightweight processing, providing bounded latency under high concurrency and large tool sets. It is a clear intelligence boundary: reasoning stays on the agent side, while the gateway performs deterministic retrieval.

3.4 Session-Aware Load Balancing

Routing requests from the same session to different backends can violate correctness (Challenge 4). At production scale, the gateway must also sustain high availability under horizontal scaling and failover, which makes session-aware routing a first-order requirement. Accordingly, the gateway provides session-aware load balancing with three objectives. **(1) Session affinity.** Requests within the same logical session are consistently routed to the same backend MCP server instance under replication. **(2) Session consistency.** Under a replicated gateway deployment, different gateway instances must apply the same session-to-backend mapping, requiring shared and consistent session routing state; thus, the system synchronizes session states across gateway instances. **(3) Transparency.** Session-aware routing is enforced within the gateway and is transparent to both MCP clients and backend servers. These designs preserve session correctness while retaining the scalability and availability benefits of horizontal scaling and failover in production deployments.

4 Implementation

We implement the gateway on Alibaba Cloud's production Layer-7 load balancing infrastructure: protocol processing and session-aware routing are implemented natively in Go (~8K LoC), the OpenAPI conversion component is compiled into a WebAssembly module loaded by the gateway runtime (~6K LoC), and authentication reuses existing L7 load balancer components. The following subsections describe how the four design blocks in §3 are realized.

4.1 Cross-Layer Protocol Translation

Unlike conventional L7 traffic, which is routed on hosts, paths, and headers. MCP routing is body-driven: tool identity and arguments are carried in JSON-RPC messages within the HTTP body, while session identifiers are transport-dependent, appearing in headers or SSE streams. The gateway therefore performs deep body or stream inspection to make tool-level routing and track session context.

Protocol conversion for API and MCP. Following the one-to-one mapping policy (§3.1), the gateway implements an automatic OpenAPI-to-MCP compilation pipeline. At tool mount time, the gateway parses the OpenAPI specification and compiles each operation into an MCP tool definition, generating MCP-compliant metadata (names, descriptions, and schemas) from OpenAPI parameters. As a result, legacy API operations become invocable through standard MCP primitives (e.g., *List Tools* and *Call Tool*). The pipeline applies a deterministic rule-mapping, compiled and constraint-checked at mount time (fail-fast); specs lacking semantics are enriched by an offline LLM pass, keeping the serving path deterministic.

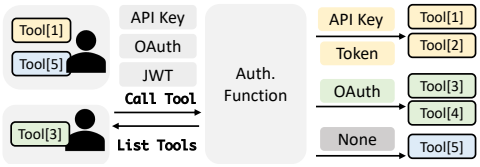


Figure 6. Fine-grained access-control function offloading

Multi-stage request routing. MCP services use HTTP+SSE or Streamable HTTP, while legacy OpenAPI services are exposed as MCP tools via protocol conversion. Upon receiving a request, the gateway resolves the execution target and transport through a staged routing pipeline (Figure A4) that **parses** the tool invocation request in both header and body, **annotates** it with backend/transport metadata, **rewrites** it into a backend operation (e.g., protocol conversion for API and MCP), and **forwards** it to the selected backend. We defer the pipeline details to Appendix D. The gateway terminates client connections and (re-)establishes backend connections according to the resolved transport, keeping the MCP client thin as transports evolve. This staged structure realizes the decoupling principle in §3.1: new transports are supported by extending individual routing stages, without changing data-layer conversion or end-to-end integration.

4.2 Authentication and Tool Access Control

As an instantiation of function offloading (§3.2), the gateway currently offloads two security functions from backends: authentication and fine-grained tool access control (Figure 6).

Authentication with ingress/egress management. Clients and backends each use heterogeneous credential schemes; pairing them directly would reintroduce an $N \times M$ integration problem at the credential level. The gateway implements a two-stage model. On ingress, it authenticates client requests and establishes a stable identity using standard methods (e.g., API keys, OAuth, and JWT). On egress, it integrates backend-specific authentication requirements by mapping the established identity to the credentials and formats expected by each backend (e.g., API keys, tokens, or OAuth-based flows). The gateway provides a security boundary by restricting backend access to gateway-managed channels at ingress.

Fine-grained tool access control. With authentication management centralized at the gateway, the gateway can enforce identity-based, tool-level authorization consistently across heterogeneous backends. Specifically, it governs *List Tools*, *Call Tool*, and the meta tool *Tool Search* (discussed in §4.3), ensuring that each authenticated user can only discover and invoke the subset of tools permitted by policy. This design exposes a tailored toolset per identity while keeping backend tool servers lightweight.

4.3 Hybrid Tool Recommendation

The gateway realizes intent-driven recommendation (§3.3) by selecting intent-relevant candidates with deterministic

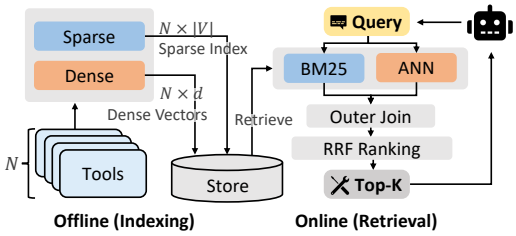


Figure 7. Hybrid tool retrieval engine

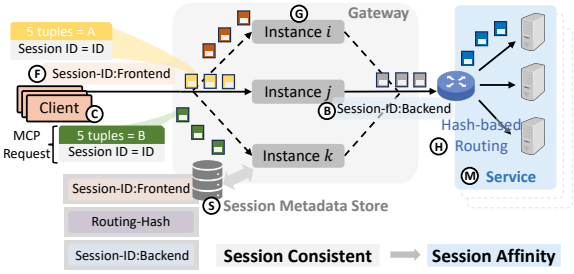


Figure 8. Session-aware request routing

execution; the host then invokes tools through the standard LLM loop over this reduced set.

Interface for tool recommendation. To realize agent-driven discovery, the recommendation capability must be invocable from the agent tool-calling loop. The original MCP *List Tools* does not support query- or constraint-based input as retrieval parameters, limiting the functionality of tool recommendation interfaces. Extending *List Tools* with additional parameters would reduce compatibility and lead to inconsistent behavior across MCP implementations. To maximize ecosystem compatibility while enabling tool recommendation, the gateway exposes a *Tool Search* interface as a standard MCP tool. This tool is always included in the tool list return by *List Tools* primitive, allowing the host to *discover tools by calling a tool*. The host submits a `query` via *Call Tool*, and receives a subset of relevant tools in response, with the same structure as the tools returned by *List Tools*.

Semantic and lexical indexing and matching. The tool retrieval engine incorporates offline indexing and online matching (Figure 7). Offline, at tool mount time, the gateway extracts textual fields (e.g., tool name and description) and constructs two complementary indexes: (1) a dense index that embeds tool texts into d-dimensional vectors with a pre-trained encoder, enhancing recall for paraphrases and fuzzy intents; (2) a sparse full-text index over the tool vocabulary, strengthening exact matching for tool names, resource identifiers, and parameter terms. At query time, the two paths retrieve candidates independently, via approximate nearest-neighbor (ANN) search on the dense index and BM25 scoring on the sparse index. The engine then outer-joins the two candidate lists and fuses them with Reciprocal Rank Fusion (RRF), scoring each tool by the sum of its reciprocal ranks with a smoothing constant, and returns the top-K tools.

4.4 Stateful Request Routing

Elastic scaling and session-aware routing. MCP servers are deployed as multiple replicas behind a service endpoint, and the gateway itself runs as multiple instances. For stateless deployments, each request can be routed independently to any healthy replica. The rest of this subsection describes how the gateway enforces session affinity and consistency (§3.4) for stateful deployments (Figure 8).

State synchronization via centralized store. A single logical session spans two identifiers: `Session-ID:Frontend`, generated by the gateway toward the client, and `Session-ID:Backend` (Ⓑ), assigned by the backend MCP Service. Any gateway instance serving the session must resolve this mapping consistently. The gateway stores the mapping in a centralized *Session Metadata Store* (Ⓢ in Figure 8).

The gateway (Ⓖ) terminates client (Ⓒ) connections and establishes backend connections to MCP Services(Ⓜ) on demand; it must construct a logical mapping between frontend and backend sessions. During session establishment, two identifiers appear: `Session-ID:Frontend`, generated by the gateway to identify the frontend session; `Session-ID:Backend` (Ⓑ), returned by the backend MCP Service to identify the backend session. When a client session is created, the gateway generates a `Session-ID:Frontend`, caches it locally, and creates a record in the Session Metadata Store (Ⓢ). When subsequent requests trigger access to a specific MCP Service, the gateway establishes the backend session on demand, obtains the `Session-ID:Backend`, and writes/updates the mapping between `Session-ID:Frontend` and `Session-ID:Backend` in the Session Metadata Store. If later requests of the same session are load-balanced to a different gateway instance, the instance can retrieve the mapping from the store and continue using the backend session, preserving session consistency during scaling events or failover, without any point-to-point synchronization among instances.

Hash-based session-affinity in segmented deployments. The gateway and MCP Services are operated and scaled independently as two tiers; the gateway cannot track instance churn inside each service. Each MCP Service exposes a single domain endpoint and implements stateless, hash-based consistent forwarding (Ⓗ) inside the service, mapping requests stably to a specific backend MCP server. Since the frontend session is established before the backend session, after generating `Session-ID:Frontend`, the gateway computes a `Routing-Hash` from it and places the hash into an HTTP request header on the *Gateway Instance -> Service* path as the key for intra-service consistent forwarding.

The gateway system uses a hash instead of propagating the original session identifier to hide the raw identifier across the gateway–compute boundary and reduce coupling. The MCP service simply forwards requests based on the Routing-Hash carried in the header without having to interpret it. The gateway stores the Routing-Hash as part of the session

metadata, together with `Session-ID:Frontend` and its corresponding `Session-ID:Backend`, in the Session Metadata Store. This allows any gateway instance that later takes over the session to reuse the same routing key, preserving session affinity and cross-gateway consistency across requests.

Long-lived connection bridging with Pub/Sub. Different transport modes impose different constraints on session consistency. For Streamable HTTP, connections are request-scoped: even with streaming, the channel terminates after the response completes, so a reply can always be returned by the gateway instance that received the request. In contrast, under HTTP+SSE, the client establishes a long-lived, instance-bound SSE channel to a specific gateway instance; all server-to-client events must be delivered through that channel. This creates a structural mismatch in a multi-instance gateway: subsequent requests carrying the same `Session-ID:Frontend` may be load-balanced to a different gateway instance that does not own the SSE return path, and thus cannot deliver responses to the client.

To resolve this mismatch, we implement a lightweight cross-instance coordination layer based on a centralized *Pub-/Sub* channel. The key idea is to explicitly separate *where a request is processed* from *where a response must be returned*. Upon receiving a request, the gateway determines whether the local instance owns the required return path; if not, it forwards either the *request* or the *response* across instances via Pub/Sub. Specifically, when the frontend uses SSE, requests are redirected to the gateway instance that holds the client-facing SSE channel; when the backend uses SSE, responses may be redirected to the instance that must return the HTTP response. This bridging is activated only when a cross-instance path discontinuity would otherwise occur; in all other cases, requests and responses are handled locally. We summarize the per-mode handling rules in Figure A6 and provide a detailed case breakdown in Appendix F.

5 Deployment

Production Scale. MCP Gateway has been deployed across five cloud regions. It supports OpenAPI 3.0/3.1, and has been validated on $O(10^4)$ OpenAPI operations across 315 Alibaba Cloud products, though currently with limited support for asynchronous and streaming. The maximum observed per-user tool mount count reaches $O(10^4)$. The gateway system has operated continuously in production for 8 months.

6 Evaluation

We answer three questions: (1) does the gateway improve agent tool selection at cloud scale (§6.2); (2) does recommendation remain accurate with bounded latency (§6.3); and (3) what overhead does the gateway add, and does it scale (§6.4)?

Table 3. Tool selection time cost, Qwen3-Plus

	Avg (ms)	P50 (ms)	P95 (ms)	P99 (ms)
Top-15	6236.96	5527.63	12399.04	15666.76
Top-30	8310.92	7683.10	14855.51	16344.22
Top-60	14364.14	14208.26	21085.03	25581.55
Baseline	55662.68	55029.86	59962.32	67936.70

Table 4. Token consumption, Qwen3-Plus

	Avg (k)	P50 (k)	P95 (k)	P99 (k)
Top-15	21.22	20.67	28.29	43.75
Top-30	39.54	39.68	56.48	62.57
Top-60	79.75	81.56	105.70	121.66
Baseline	505.86	507.29	511.38	512.56

6.1 Evaluation Setup

Environment. Experiments data are collected or conducted from the production regions above. Gateway pods (2 cores, 4 GB each) run on Intel Xeon Platinum 8369B servers; the tool retrieval engine runs separately on 4-core, 16 GB, 50 GB-storage machines and scales independently.

Workloads. *Alibaba Cloud*: up to 3,616 tools from our production catalog and 4 production traces over VPC, ECS and SLB services, representing typical user traffic. *ToolBench* [47] (up to 640 tools): a public benchmark for comparability.

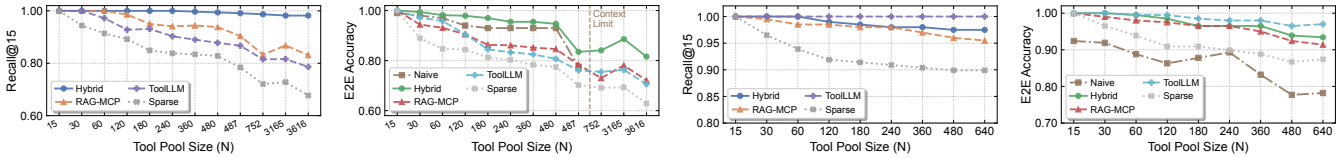
Baselines. (1) *RAG-MCP* [20]: dense embeddings only; (2) *ToolLLM* [47]: retriever fine-tuned on ToolBench; (3) *BM25*: sparse retrieval; (4) *Naive*: no retrieval, all tools inlined.

Protocol and Metrics. The retriever returns Top-15, from which the LLM selects one. We report *Recall@15* (ground truth in candidates, §6.3), *selection accuracy* (LLM picks the target tool, §6.2), and completion time and token usage (§6.2).

6.2 Agent Tool Selection Performance

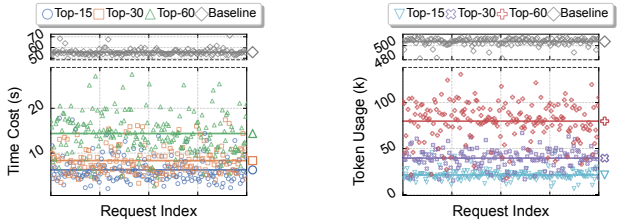
Tool selection accuracy on cloud-scale tools. The retrieval-then-selection pipeline consistently improves over the expose-all approach. As shown in Figure 9b and Figure 9d, the Naive baseline degrades as the tool pool grows: on ToolBench, its selection accuracy drops from 92.4% at N=15 to 78.2% at N=640, falling below all retrieval-based methods; on Alibaba Cloud tools, it exceeds the LLM context window at $N \geq 752$ and becomes infeasible. In contrast, retrieval-based methods maintain stable accuracy: at N=3,616, Hybrid achieves 81.6% versus 72.1% for RAG-MCP and 70.6% for ToolLLM. Thus, pre-filtering to a compact candidate set before LLM selection is both necessary for scalability and beneficial for accuracy.

Tool selection latency and token cost. Figure 10 shows that the end-to-end agent task completion evaluation follows the Agent–Gateway–Tools path and uses representative categories of production user traffic (VPC, ECS and SLB tools). The gateway hosts $O(1k)$ tools for cloud resource operations. In the baseline setting, the agent selects tools from 372 tools, since the baseline mounts and exposes all tools and cannot scale to mounting $O(1k)$ tools. For each request, the agent selects tools from the recommendation list (Top-15, 30, 60), and then issues tool calls through the gateway. As shown in Table 3, on average, Top-15 reduces time cost from 55.7



(a) Production: Recall@15 **(b)** Production: Selection Accuracy **(c)** ToolBench: Recall@15 **(d)** ToolBench: Selection Accuracy

Figure 9. Retrieval comparison across methods on production cloud tools (a–b) and ToolBench (c–d), Qwen3.6-plus



(a) Time cost **(b)** Token usage

Figure 10. LLM agent tool selection performance

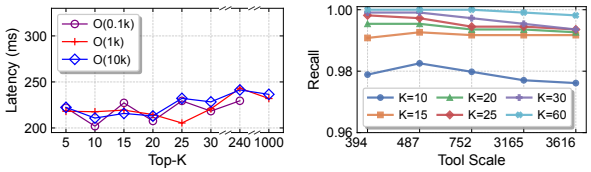


Figure 11. Retrieval latency v.s. Tool scale

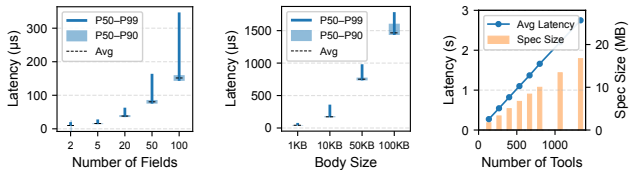
Figure 12. Top-K recall v.s. Tool scale

s (baseline) to 6.2 s, a reduction of 8.9×. Table 4 indicates that token consumption decreases from 505.86k to 21.22k, a reduction of 23.8×. On tail latency, Top-15 reduces P95 from 60.0 s to 12.4 s and P99 from 67.9 s to 15.7 s.[2]

6.3 Hybrid Tool Recommendation

Latency and accuracy at scale. Figure 11 shows that recommendation latency remains stable from $O(0.1k)$ to $O(10k)$ tools. Across recommendation set sizes, including Top-5, Top-10, Top-15, and up to Top-1000, vector retrieval and approximate similarity matching complete within 202–243 ms, delivering the bounded latency promised by the gateway-side deterministic design (§3.3). As illustrated in Figure 12 and Appendix H, accuracy remains high as the tool set grows. With the mounted tool set increasing from 394 to 3616 tools, Top-15 retrieval achieves over 98% accuracy. It further shows a sharp accuracy gain when expanding the recommendation set from Top-10 to Top-15, indicating Top-15 as an effective operating point for tool recommendation. We discuss the K selection strategy in Appendix E. Beyond these results, our production traces reveal an additional class of *weak-semantic* queries (e.g., ID-only requests). The semantic signal of this request is too sparse for retrieval. We report this observation and production optimization as (Lesson 4) in §7.

[2]The reported latency measures the interval from the agent initiating tool recommendation to the gateway receiving the subsequent tool-call request.



(a) Field count **(b)** Body size **(c)** Offline

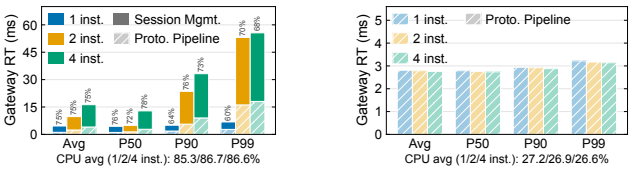
Figure 13. API conversion overhead: online request conversion (a, b) and offline tool generation (c).

Comparison with retrieval baselines. Figure 9 presents the results. On ToolBench (Figures 9c and 9d), ToolLLM achieves near-perfect recall across all pool sizes, since ToolBench is the corpus on which its retriever was fine-tuned. However, this advantage does not transfer. On Alibaba Cloud tools (Figures 9a and 9b), ToolLLM's recall drops sharply as the tool pool grows, falling to 78.6% at N=3,616, below even its performance on ToolBench at equivalent scale. This confirms that fine-tuning on a fixed benchmark does not generalize to real-world cloud tool catalogs, where tool names encode domain-specific identifiers and descriptions follow cloud-specific conventions. In production, where the tool is diverse and changes continuously, such re-tuning is impractical because every update requires data generation, retraining, and redeployment.

Without any fine-tuning, the gateway's hybrid retrieval consistently outperforms all baselines on Alibaba Cloud tools. At N=3,616, Hybrid maintains 98.2% Recall@15 while RAG-MCP (dense-only) drops to 83.1%, ToolLLM to 78.6%, and Sparse (BM25) to 67.7%. The advantage of combining semantic and lexical matching is most pronounced at large scale. Dense embeddings capture intent-level paraphrases that sparse matching misses, while BM25 preserves exact matches on resource identifiers and parameter names that dense encoders often overlook. On ToolBench, where tool descriptions are more generic, the hybrid approach still matches or slightly outperforms RAG-MCP (97.5% vs. 95.5% at N=640), confirming that adding sparse signals does not hurt when lexical cues are less informative.

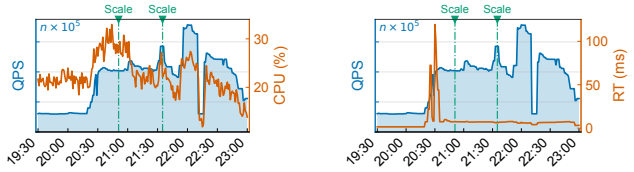
6.4 Gateway Overhead and Scalability

API conversion overhead is negligible. Figure 13 reports the latency of converting an MCP tool-call request into the corresponding API invocation on the gateway. Payloads vary from 2–100 JSON fields and 1–100KB bodies. As shown in Figure 13a, for typical requests with fewer than 20 fields,



(a) Stateful gateway **(b)** Stateless gateway

Figure 14. Gateway RT breakdown vs. # instances



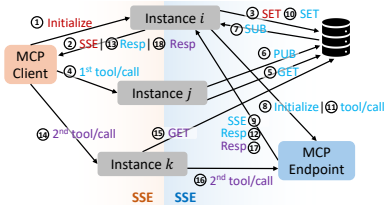
(a) QPS & CPU usage **(b)** QPS & RT

Figure 15. The scalability of the gateway system

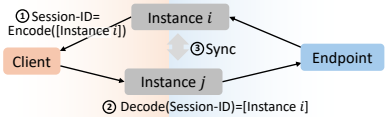
the median (P50) conversion latency stays below 35 μs. Even for the largest case with 100 fields, the P50 latency is only 143 μs. Figure 13b shows that the conversion scales linearly with body size. Figure 13c reports the offline conversion that generates MCP tools from API specs. The conversion time grows linearly with the number of tools. Even for a large spec of 16.7 MB, generating 1,334 tools takes only 2.75 s. The offline conversion runs once per API registration and off the critical path, making its cost easily amortized.

Session-aware routing dominates per-request cost. Figure 14a shows that stateful routing adds substantial overhead, even before horizontal scale-out. With a single gateway instance at ~ 100 QPS, the mean RT is 4.59 ms with P50/P90/P99 of 4.41/5.02/6.81 ms. Session Management—session-affinity lookup and routing—dominates the cost, contributing 3.45 ms on average (75% of mean RT) and 4.07 ms at P99, while the Protocol Pipeline (transport parsing and MCP dispatch) costs 1.14 ms. At this load, the stateful instance already consumes 85.3% CPU, versus 27.2% for the stateless one.

Cross-instance misses amplify latency. As the gateway scales out, this overhead grows. With four instances, mean RT rises to 16.26 ms and P99 reaches 55.61 ms, about 8× the single-instance P99, with Session Management P99 growing from 4.07 ms to 37.55 ms. The increase comes mainly from session data misses. When the upstream load balancer routes a session to an instance other than the one that holds it, the gateway must fetch session metadata from the remote store. The hit–miss split is visible in the percentiles: with two instances, P50 remains near the single-instance level (4.94 ms) while P90 jumps to 23.58 ms; with four instances, the miss probability rises further and even P50 climbs to 12.87 ms. The stateless gateway (Figure 14b) confirms this: across instance counts, mean RT stays within 2.76–2.81 ms, P99 below 3.3 ms, and CPU around 27%. While the centralized store enables stateless scale-out, it places the latency bottleneck on the miss path, especially for MCP traffic, where we observe a 2× read/write amplification to the store (§7).



(a) Centralized architecture



(b) Session-ID encoding architecture

Figure 16. From centralized management to encoding.

Elastic scale-out with stable RT. Figure 15 shows that the gateway maintains stable response time (RT) around 7 ms. As load increases, the gateway scales out on CPU pressure: at 20:50:52, as traffic rises around 20:30 and per-instance CPU increases from ~20% to ~30% (Figure 15a), the fleet scales and drives CPU back toward the ~20% steady level without perturbing RT. A second scale-out occurs at 21:35:00 under a QPS-driven CPU increase, again with stable RT. This stability indicates that the design does not require point-to-point state synchronization among gateway instances (§3.4), allowing new instances to join and absorb load without adding extra coordination. Note that the RT spike around 20:30:00 is due to a backend scale-out failure causing request backlog on the gateway, rather than a limitation of the gateway scaling.

7 Lessons Learned from Deployments

Our production deployment reveals challenges that do not emerge in small-scale prototypes, driven by two forms of scale. At *traffic* scale, session management, routing, and gateway–backend connections must remain efficient across distributed instances (1&2). At *tool* scale, the gateway must recommend and compose tools from a large, evolving catalog for queries that are often under-specified (3&4).

Lesson 1: Coordination for session consistency. In production, we observed that in the scenario where both frontend and backend sessions use SSE, a session is initialized on one gateway instance while subsequent tool calls are routed to others (Figure 16a). To preserve correctness, the deployment relies on a centralized coordination module (e.g., a session store such as Redis plus Pub/Sub). However, the coordination is non-trivial. In the worst case, handling only three MCP requests (one initialization and two *Call Tool* requests) triggers six interactions with the centralized module (③⑤⑥⑦⑩⑮), making it both a performance bottleneck and a reliability risk point. Moreover, it introduces a hard external dependency, increasing cost and coupling the gateway SLA to the external store and messaging system. An alternative is fully distributed synchronization among gateway

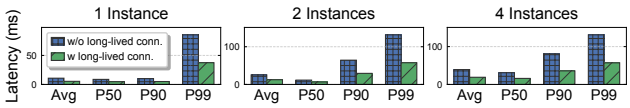


Figure 17. Latency w/o and with long-lived connection

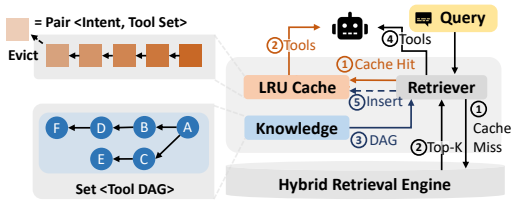


Figure 18. Semantic cache

instances, but it is complex and can generate substantial synchronization traffic, especially under elastic scaling.

However, we observe a property specific to MCP: the gateway assigned `Session-ID` is carried by the MCP client on every request. This makes the session identifier an effective control signal for routing and coordination. Our system is evolving toward a *Session-ID encoding* approach. As shown in Figure 16b, during session establishment, the gateway encodes an owner hint (e.g., an instance identity or a routable locator) into the `Session-ID` returned to the client. When a later request of the same session arrives at a different gateway instance, the receiving instance decodes the `Session-ID` to locate the owner and coordinates with that instance if needed. Despite being fully distributed, this design keeps synchronization overhead bounded.

Lesson 1: *Centralized session stores become the bottleneck at scale; exploit identifiers the protocol already carries—encode ownership into the Session-ID.*

Lesson 2: Long-lived gateway–backend connection. A naive design was initially adopted to simplify implementation: the gateway did not maintain persistent connections to the backend. Each tool call triggered a full connect–initialize–invoke sequence, followed by connection teardown. Deployment measurements show that this design introduces substantial user-perceived latency. From the client perspective, a tool call is a single request, but the naive design expands it into three gateway–backend interactions. As shown in Figure 17, without long-lived gateway–backend connections, the 4 instance tail latency exceeds 100 ms. The resulting lesson is that, when the backend requires long-lived connections, the gateway should absorb this complexity to avoid multi-round-trip overhead on the critical path. After the redesign, long-lived gateway–backend connections reduce mean/P90/P99 latency by 51.7%/55.7%/56.5%.

Lesson 2: *When backends require long-lived connections, the gateway should absorb that complexity.*

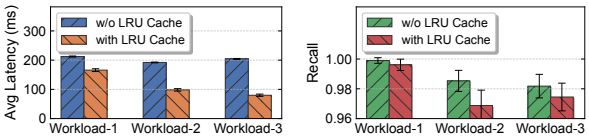


Figure 19. LRU caching on latency with varied production trace workloads (a); and recall with different query sets (b)

Table 5. Knowledge Cache evaluation. Prereq. Completeness measures the fraction of queries whose *all* prerequisite tools are retrieved. Extra RTTs and Tokens are per-query averages incurred by iterative LLM discovery.

Set	Method	Compl.(%)	+RTTs	+Tokens
Set-1 (394 tools)	Baseline	29.1	0	0
	Discovery	73.3	1.03	114
	Know. Cache	**100.0**	**0**	**0**
Set-2 (3616 tools)	Baseline	24.5	0	0
	Discovery	56.1	1.00	110
	Know. Cache	**100.0**	**0**	**0**

Lesson 3: Semantic caching for tool recommendation. Users often describe an *end goal* in natural language, while successful execution often requires a tool-calling chain with strict prerequisites. For instance, creating an Application Load Balancer typically requires creating and configuring a VPC and VSwitch first. This exposes a limitation of semantic-similarity-only tool recommendation: it may retrieve the target tool but miss its prerequisite tools. When prerequisites are missing, the agent-side LLM must discover them at inference time by issuing additional retrieval-tool calls, each incurring an extra LLM round-trip, added latency, and token overhead. As dependency chains grow deeper, this iterative discovery compounds: each step has an independent success rate, so the probability of recovering the full chain decays exponentially with depth.

(1) Knowledge Cache. To eliminate this inference-time overhead, the gateway system introduces a *Knowledge Cache* that precomputes prerequisite chains from the schema (Figure 18). At indexing time, the gateway analyzes foreign-key dependencies across the tool corpus and builds tool chain DAGs via topological sort. At query time, after the retrieval engine returns candidate tools, the Retriever looks up each candidate's prerequisite chain in the Knowledge Cache and merges the prerequisite tools into the final result, so the agent receives an executable chain in one shot, no additional LLM round-trips needed. As shown in Table 5, the Knowledge Cache achieves 100% prerequisite completeness on both sets with zero extra RTTs and zero extra tokens, while iterative LLM discovery requires ~1 extra round-trip per query yet still drops to 0% completeness on long chains (Table A1 in Appendix I).

(2) LRU Cache. We observed that online queries exhibit strong semantic skewness, where a small set of similar queries dominates. We cluster the query intents in a representative

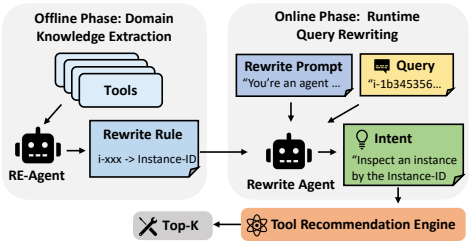


Figure 20. Rewrite-Agent and RE-Agent workflow

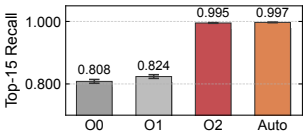


Figure 21. Query rewrite optimizations (on Trace-3)

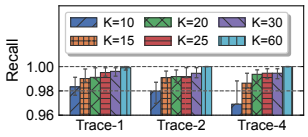


Figure 22. Top-K recall v.s. Traces

production trace into 224 clusters, and the top 10 clusters account for 80.2% of the traffic (Figure A7). The gateway adds an LRU *semantic cache* for ⟨*Query*, *Tool Set*⟩ pairs to bypass retrieval on cache hits. Even a small cache (e.g., 64 entries) reduces average end-to-end latency by up to 60% under varied workloads without sacrificing recall (only related to the semantic characteristics of query workloads) (Figure 19a, Figure 19b).

Lesson 3: *Semantic alone cannot recover tool dependencies; precompute prerequisite chains offline and cache skewed intents for low latency.*

Lesson 4: Query rewriting for domain-specific, underspecified intents. In practice, many user queries are underspecified. A common case is an *ID-only* query that contains just an identifier string (e.g., an instance or user ID). Such inputs carry little textual signal, while tool descriptions are in natural language. This mismatch destabilizes retrieval-based tool recommendation, as the system cannot reliably infer the target entity or intended action from an identifier alone. To address the mismatch, as shown in Figure 21, a sequence of optimizations is introduced, improving Top-15 Recall from 80.8% (O0) and 82.4% (O1) to 99.5% (O2), and further to 99.7%.

We first implement an LLM-based *Rewrite-Agent* to rewrite the input into a normalized, domain-aware intent before tool recommendation (Figure 20). Given a fixed prompt template, it converts the raw query into an *Intent* that better matches tool descriptions, and the retrieval engine then performs Top-K tools recommendation based on the intent (O1 in Figure 21). The key is *entity normalization*: mapping identifiers and other non-linguistic tokens to an abstract entity type used in tool descriptions (e.g., rewriting an instance ID into an intent targeting a *compute instance*).

However, a general Rewrite-Agent often performs poorly for ID-only queries because it lacks domain knowledge. Identifier formats, object taxonomies, and naming conventions vary across services, and the model may map an identifier to the wrong entity type, which directly degrades recall. To address this, we first encode domain knowledge as a set of rewrite rules in the Rewrite prompt. These rules map common identifier patterns to entity types, steering the Rewrite-Agent to produce intents that align with tool descriptions. With rule-guided rewriting, the Top-15 recall of tool recommendation increases to 99.5% (O2 in Figure 21).

Maintaining such rules manually does not scale. We therefore design an *RE-Agent* (Rule-extraction agent, Figure 20) that automatically extracts rewrite rules from tool descriptions. Each rule maps an identifier pattern to an entity type. We inject the extracted rules into the Rewrite prompt to guide normalization, and then let the Rewrite-Agent produce the final Intent. This automated, rule-guided pipeline improves tool recommendation for ID-only and weak-semantic queries and maintains Top-15 recall of 99.7% (Auto in Figure 21).

Using the final configuration (Auto), we analyze multiple production traces where the gateway serves $O(1k)$ cloud operation tools (including MCP tools and API operations), and agents rely on tool recommendation for tool discovery during task execution. Figure 22 shows consistently high accuracy: Top-10 recall exceeds 96%, while Top-15 and larger sets exceed 98%.

Lesson 4: *Under-specified queries need domain knowledge, and that knowledge can be extracted automatically.*

8 Related Work

L7 Load Balancers and Gateways. L7 gateways and load balancers are widely deployed in cloud environments. Prior systems work has studied L7 load balancing and gateway designs in production settings [41, 49, 62]. There are also open-source L7 gateways and load balancers (e.g., NGINX and Envoy) [7–10]. L7 load balancers and gateways primarily make routing decisions based on HTTP metadata (e.g., host, path, and headers) and typically treat request bodies as opaque. As a result, they do not natively support inspecting nested JSON-RPC payloads required by MCP, and they cannot directly differentiate tool-invocation requests or infer protocol-level capabilities without non-trivial extensions.

LLM Agent Infrastructures. Agent infrastructure work studies efficient execution and serving of LLM-based applications, including serving systems for LLM inference [28, 46, 69] and scheduling inference engines for agentic programs [11, 14, 29, 31, 34, 53]. Orthogonal to our focus, Cortex [48] reduces latency and cost for remote knowledge/data access in LLM agents via semantic-aware caching, whereas our gateway system focuses on scalable tool access, multi-version MCP transport support, and routing across heterogeneous tool servers.

Another line of work focuses on agent architectures, such as memory management [40, 65, 70], reasoning frameworks [61,

67, 68], and agent runtimes [37], to improve task performance and robustness. The gateway targets the tool-calling infrastructure, providing efficient and scalable tool access. Other studies improve tool selection and argument generation [42, 47, 50] and enhance tool-calling reliability under large tool sets [20]. Recent MCP-related studies examine protocol adoption, reliability, and security [26].

Both open-source systems [2, 17, 38] and commercial systems (e.g., AgentCore Gateway [1]) support MCP integration to enable standardized tool access; however, each addresses only a subset of the challenges in §2.3. To the best of our knowledge, our system is the first that bridges incompatible MCP transport variants while preserving session consistency across replicated gateway instances, and the first to report design and measurements at a cloud-scale deployment.

9 Conclusion

Deploying MCP services at cloud scale is not a drop-in change, and traditional L7 load balancers cannot be directly applied to MCP. This motivates the a gateway system. The protocol adaptation module translates across multiple protocols and connects agents to large-scale legacy API servers, addressing tool provider-side integration. Gateway-side function offloading centralizes fine-grained access control via ingress and egress authentication and reduces the burden on MCP developers. Tool recommendation provides efficient and scalable tool selection, reducing agent tool selection time and token usage, and addressing the agent-side challenge of accessing massive tools. Session-aware routing enables replicated MCP backends and elastic gateway scale-out for stateful deployments, while the gateway achieves low processing latency by supporting long-lived sessions. Our results show that the gateway enables massive tool access and scales agent-side tool access in production.

References

[1] AWS Bedrock AgentCore. 2026. *AgentCore Gateway.* https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/gateway.html Accessed: 2026-01-06.

[2] agentgateway authors. 2026. *agentgateway: Next Generation Agentic Proxy for AI Agents and MCP servers.* https://agentgateway.dev/ Accessed: 2026-06-10.

[3] Alibaba Cloud. 2025. *Alibaba Cloud API Documentation.* https://api.alibabacloud.com/document Accessed: 2025-12-17.

[4] Anthropic. 2025. *Claude Opus 4.5.* https://www.anthropic.com/news/claude-opus-4-5 Accessed: 2026-01-06.

[5] Anthropic. 2025. *Introducing Claude Opus 4.5.* https://www.anthropic.com/news/claude-opus-4-5 Accessed: 2026-01-06.

[6] Anthropic. 2026. *Introducing the Model Context Protocol.* https://www.anthropic.com/news/model-context-protocol Accessed: 2026-01-06.

[7] Apache APISIX Authors. 2026. *Apache APISIX Documentation.* https://apisix.apache.org/docs/apisix/getting-started/README/ Accessed: 2026-01-06.

[8] Envoy Project Authors. 2026. *Envoy Documentation.* https://www.envoyproxy.io/docs/envoy/latest/ Accessed: 2026-01-06.

[9] HAProxy Project Authors. 2026. *HAProxy Documentation.* https://docs.haproxy.org/ Accessed: 2026-01-06.

[10] NGINX Project Authors. 2026. *NGINX Documentation.* https://nginx.org/en/docs/index.html Accessed: 2026-01-06.

[11] Zhuohang Bian, Feiyang Wu, Teng Ma, and Youwei Zhuo. 2025. Tokencake: A KV-Cache-centric Serving Framework for LLM-based Multi-Agent Applications. *arXiv preprint arXiv:2510.18586* (2025).

[12] Thierry Blankenstein, Jialin Yu, Zixuan Li, Vassilis Plachouras, Sunando Sengupta, Philip Torr, Yarin Gal, Alasdair Paren, and Adel Bibi. 2025. BiasBusters: Uncovering and Mitigating Tool Selection Bias in Large Language Models. *arXiv preprint arXiv:2510.00307* (2025).

[13] Alan Chan, Kevin Wei, Sihao Huang, Nitarshan Rajkumar, Elija Perrier, Seth Lazar, Gillian K Hadfield, and Markus Anderljung. 2025. Infrastructure for ai agents. *arXiv preprint arXiv:2501.10114* (2025).

[14] Jinyuan Chen, Jiuchen Shi, Quan Chen, and Minyi Guo. 2025. Kairos: Low-latency multi-agent serving with shared llms and excessive loads in the public cloud. *arXiv preprint arXiv:2508.06948* (2025).

[15] LangChain Contributors. 2025. *LangChain: Building applications with LLMs through composability.* https://www.langchain.com Accessed: 2025-11-19.

[16] Model Context Protocol Contributors. 2026. *MCP Specification (Protocol Revision: 2024-11-05).* https://modelcontextprotocol.io/specification/2024-11-05 Accessed: 2026-01-06.

[17] MCP Hub Contributors. 2026. *MCP Hub.* https://github.com/ravitemer/mcp-hub Accessed: 2026-01-06.

[18] DeepSeek. 2026. *DeepSeek API Docs.* https://api-docs.deepseek.com/quick_start/pricing Accessed: 2026-01-06.

[19] Meihao Fan, Ju Fan, Nan Tang, Lei Cao, Guoliang Li, and Xiaoyong Du. 2024. AutoPrep: Natural Language Question-Aware Data Preparation with a Multi-Agent Framework. *arXiv preprint arXiv:2412.10422* (2024).

[20] Tiantian Gan and Qiyao Sun. 2025. Rag-mcp: Mitigating prompt bloat in llm tool selection via retrieval-augmented generation. *arXiv preprint arXiv:2505.03275* (2025).

[21] Dawei Gao, Zitao Li, Xuchen Pan, Weirui Kuang, Zhijian Ma, Bingchen Qian, Fei Wei, Wenhao Zhang, Yuexiang Xie, Daoyuan Chen, et al. 2024. Agentscope: A flexible yet robust multi-agent platform. *arXiv preprint arXiv:2402.14034* (2024).

[22] Dawei Gao, Zitao Li, Yuexiang Xie, Weirui Kuang, Liuyi Yao, Bingchen Qian, Zhijian Ma, Yue Cui, Haohao Luo, Shen Li, et al. 2025. AgentScope 1.0: A Developer-Centric Framework for Building Agentic Applications. *arXiv preprint arXiv:2508.16279* (2025).

[23] Google Gemini. 2026. *Gemini API Documents.* https://ai.google.dev/gemini-api/docs/models Accessed: 2026-01-06.

[24] In Gim, Zhiyao Ma, Seung-seob Lee, and Lin Zhong. 2025. Pie: A programmable serving system for emerging llm applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 415–430.

[25] Hechuan Guo, Yongle Hao, Yue Zhang, Minghui Xu, Peizhuo Lv, Jiezhi Chen, and Xiuzhen Cheng. 2025. A measurement study of model context protocol ecosystem. *arXiv preprint arXiv:2509.25292* (2025).

[26] Yongjian Guo, Puzhuo Liu, Wanlun Ma, Zehang Deng, Xiaogang Zhu, Peng Di, Xi Xiao, and Sheng Wen. 2025. Systematic analysis of mcp security. *arXiv preprint arXiv:2508.12538* (2025).

[27] Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. 2025. Model context protocol (mcp): Landscape, security threats, and future research directions. *arXiv preprint arXiv:2503.23278* (2025).

[28] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*. 611–626.

[29] Marco Laju, Donghyun Son, Saurabh Agarwal, Nitin Kedia, Myungjin Lee, Jayanth Srinivasa, and Aditya Akella. 2026. Nalar: An agent serving framework. *arXiv preprint arXiv:2601.05109* (2026).

[30] Guoliang Li, Xuanhe Zhou, and Xinyang Zhao. 2024. Llm for data management. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4213–4216.

[31] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient serving of {LLM-based} applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 929–945.

[32] Samuel Lin, Jiawei Zhou, and Minlan Yu. 2025. An LLM-based Agentic Framework for Accessible NetworkControl. *ACM SIGMETRICS Performance Evaluation Review* 53, 2 (2025), 15–20.

[33] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics* 12 (2024), 157–173.

[34] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E Gonzalez, et al. 2025. Autellix: An efficient serving engine for llm agents as general programs. *arXiv preprint arXiv:2502.13965* (2025).

[35] MCP Market. 2026. *MCP Market*. https://mcpmarket.com/ Accessed: 2026-01-06.

[36] MCP.so. 2026. *MCP.so*. https://mcp.so/ Accessed: 2026-01-06.

[37] Kai Mei, Xi Zhu, Wujiang Xu, Wenyue Hua, Mingyu Jin, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. 2024. Aios: Llm agent operating system. *arXiv preprint arXiv:2403.16971* (2024).

[38] Microsoft. 2026. *Microsoft MCP Gateway*. https://microsoft.github.io/mcp-gateway/ Accessed: 2026-01-06.

[39] Model Context Protocol. 2025. *What is the Model Context Protocol (MCP)?* https://modelcontextprotocol.io/ Accessed: 2025-12-17.

[40] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. 2023. MemGPT: Towards LLMs as Operating Systems. *arXiv preprint arXiv:2310.08560* (2023).

[41] Tian Pan, Enge Song, Yueshang Zuo, Shaokai Zhang, Yang Song, Jiangu Zhao, Wengang Hou, Jianyuan Lu, Xiaoqing Sun, Shize Zhang, et al. 2025. Hermes: Enhancing Layer-7 Cloud Load Balancers with Userspace-Directed I/O Event Notification. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 363–380.

[42] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems* 37 (2024), 126544–126565.

[43] Alibaba Cloud Bailian Platform. 2026. *Bailian MCP Market*. https://bailian.console.aliyun.com/cn-beijing/?tab=mcp#/mcp-market Accessed: 2026-01-06.

[44] OpenAI Platform. 2026. *GPT-5.2 Model Details*. https://platform.openai.com/docs/models/gpt-5.2 Accessed: 2026-01-06.

[45] Zhangcheng Qiang, Weiqing Wang, and Kerry Taylor. 2024. Agent-OM: Leveraging LLM Agents for Ontology Matching. *Proceedings of the VLDB Endowment* 18, 3 (2024), 516–529.

[46] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 155–170.

[47] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789* (2023).

[48] Chaoyi Ruan, Chao Bi, Kaiwen Zheng, Ziji Shi, Xinyi Wan, and Jialin Li. 2026. Cortex: Achieving {Low-Latency},{Cost-Efficient} Remote Data Access For {LLM} via {Semantic-Aware} Knowledge Caching. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 2407–2421.

[49] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. 2023. {ServiceRouter}: Hyperscale and minimal cost service mesh at meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 969–985.

[50] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2023), 68539–68551.

[51] Souhaila Serbout and Cesare Pautasso. 2024. APIstic: A large collection of OpenAPI metrics. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 265–277.

[52] Swagger. 2026. *Swagger RESTful API Documentation Specification*. https://docs.swagger.io/spec.html Accessed: 2026-01-06.

[53] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. 2025. Towards end-to-end optimization of llm-based applications with ayo. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1302–1316.

[54] Qwen Team. 2026. *Qwen Plus Model Details*. https://bailian.console.aliyun.com/?tab=model#/model-market/detail/qwen-plus-2025-12-01 Accessed: 2026-01-06.

[55] Qwen Team. 2026. *Qwen3 Max Model Details*. https://bailian.console.aliyun.com/?tab=model#/model-market/detail/qwen3-max-2025-09-23 Accessed: 2026-01-06.

[56] Chenxu Wang, Xumiao Zhang, Runwei Lu, Xianshang Lin, Xuan Zeng, Xinlei Zhang, Zhe An, Gongwei Wu, Jiaqi Gao, Chen Tian, et al. 2025. Towards llm-based failure localization in production-scale networks. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 496–511.

[57] Haopei Wang, Anubhavnidhi Abhashkumar, Changyu Lin, Tianrong Zhang, Xiaoming Gu, Ning Ma, Chang Wu, Songlin Liu, Wei Zhou, Yongbin Dong, et al. 2024. {NetAssistant}: Dialogue based network diagnosis in data center networks. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2011–2024.

[58] Shujun Wang, Yongqiang Tian, and Dengcheng He. 2023. gDoc: Automatic generation of structured API documentation. In *Companion Proceedings of the ACM Web Conference 2023*. 53–56.

[59] Shujun Wang, Yongqiang Tian, and Dengcheng He. 2023. Improving API Documentation Comprehensibility via Continuous Optimization and Multilingual SDK. *arXiv preprint arXiv:2303.13828* (2023).

[60] Zhaodong Wang, Samuel Lin, Guanqing Yan, Soudeh Ghorbani, Minlan Yu, Jiawei Zhou, Nathan Hu, Lopa Baruah, Sam Peters, Srikanth Kamath, et al. 2025. Intent-driven network management with multi-agent LLMs: The confucius framework. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 347–362.

[61] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[62] Ziqi Wei, Zhiqiang Wang, Qing Li, Yuan Yang, Cheng Luo, Fuyu Wang, Yong Jiang, Sijie Yang, and Zhenhui Yuan. 2024. {QDSR}: Accelerating Layer-7 Load Balancing by Direct Server Return with {QUIC}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 715–730.

[63] Georg Wölflein, Dyke Ferber, Daniel Truhn, Ognjen Arandjelovic, and Jakob Nikolas Kather. 2025. Llm agents making agent tools. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 26092–26130.

[64] Duo Wu, Xianda Wang, Yaqi Qiao, Zhi Wang, Junchen Jiang, Shuguang Cui, and Fangxin Wang. 2024. Netllm: Adapting large language models for networking. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 661–678.

[65] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110* (2025).

[66] Yingxuan Yang, Huacan Chai, Yuanyi Song, Siyuan Qi, Muning Wen, Ning Li, Junwei Liao, Haoyi Hu, Jianghao Lin, Gaowei Chang, et al. 2025. A survey of ai agent protocols. *arXiv preprint arXiv:2504.16736* (2025).

[67] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems* 36 (2023), 11809–11822.

[68] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

[69] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

[70] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2024. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 19724–19731.

Appendix

A Additional Lessons

Lesson 5: DNS locality–based VIP selection for multi-AZ Failover. To improve multi-region and multi-AZ resilience on the gateway -> MCP service data path, we implement a DNS-locality–based backend selection mechanism. An MCP Service exposes a single domain endpoint, while provisioning multiple Virtual IP (VIP) ingress points across Availability Zones. For each connection, the gateway resolves the service domain and selects the closest VIP based on DNS locality. This design provides fast failover by allowing traffic to shift to alternative VIPs when an AZ/VIP becomes unavailable, reduces cross-AZ/region hops to lower latency and tail jitter, and decouples operations by making scaling, migration, and traffic switching transparent to upstream components.

Lesson 6: Agent memory under dynamic tooling. When the set of available tools changes often, agents can suffer from memory contamination: they keep past tool calling history, or past tool plans in memory and then reuse them in a new task. This can cause tool hallucinations, where the agent calls a tool that is no longer available or uses the wrong arguments, even though the tool recommendation system returns a new Top-K tools for the current query request. A simple mitigation is a Controller–Worker design: the Controller stores stable user context and makes a high-level plan but does not keep detailed tool schemas, while a short-lived Worker mounts only the current Top-K tools, executes the calls, returns results, and is then discarded so stale tool traces do not carry over to the next task. In our evaluation, the Controller can iterate tool planning and discovery. When tool recommendation fails, or when the Worker reports that additional tools are needed, the Controller issues another round of tool recommendation and re-dispatches a Worker with the new Top-K tools. This process repeats until the required tool is found or a maximum number of iterations is reached, at which point the task is marked as failed. This iterative workflow improves the agent's task success rate.

B API Services Analysis

To quantify the scale of legacy OpenAPI infrastructures, we analyze the API specifications of a major cloud provider and summarize the number of OpenAPI operations by service category in Figure A1. The provider exposes an API surface on the order of 10^4 distinct operations: in total, 319 services define 23,284 OpenAPI operations spanning 18 categories.

Figure A2 further shows the distribution of operations across services. The distribution is highly skewed: the largest service defines 921 operations, whereas the smallest defines only 1. On average, each service provides 73 operations.

C MCP Interaction Workflow

The MCP host, i.e., an LLM application or agent, follows a one-client-per-server model: it instantiates and maintains a dedicated MCP client instance for each MCP server it connects to.

As illustrated in Figure A3, from the tool-calling perspective, the interaction begins with connection establishment and initialization. The client performs tool capability discovery by invoking the *List Tools* primitive to obtain the server's complete list of tool definitions (⓪). Typically, the tool list is cached on the host. Upon receiving a user query (①), the MCP host constructs a composite prompt that includes both the query and the discovered tool specifications. The specifications for all discovered tools, i.e., their names, descriptions, and parameter schemas, are serialized within the composite prompt. Leveraging the reasoning capabilities of the LLM, the host invokes the LLM to select the appropriate tools (②) and generates structured arguments (③) for the selected tools. The host parses the structured tool invocation arguments returned by the LLM and prepares to initiate the tool call via the client. To reduce execution risk, the host may request user authorization to execute the target tools (④). Once approved, the client issues a Tool Call primitive as a JSON-RPC request (⑤), triggering local tool execution on the MCP Server (⑥–⑦). The server returns the tool execution output to the client (⑧), and the host appends it to the ongoing LLM interaction context (⑨). The LLM then uses the updated context to determine whether the user request has been satisfied; if not, the agent repeats steps ②–⑨ until completion, at which point the LLM generates the final response (⑩).

D Request Routing Pipeline Details

Tool identifier conventions and mount-time mappings. To ensure global uniqueness across mounted backends, the gateway exposes each tool under a namespaced identifier (e.g., `service-name::tool-name`). At mount time, the gateway builds a mapping from exposed tools to their corresponding backend services, including backend type (OpenAPI vs. MCP) and the supported transport (HTTP, HTTP+SSE, or Streamable HTTP). This mapping is used at runtime to resolve the execution target for each *Call Tool* request.

Multi-stage routing pipeline. Upon receiving a *Call Tool* request, the gateway routes it via four stages:

- **Parse.** Parse data-layer fields from the JSON-RPC body (tool identifier and arguments), and extract request context such as streaming mode. In stateful deployments, the gateway also recovers session context from the transport. Under HTTP+SSE, the backend assigns the session identifier during initialization and delivers it in the SSE event stream; the gateway inspects SSE event payloads to extract the session identifier for subsequent session-aware forwarding.
- **Annotate.** Resolve the target backend service using mount-time mappings, and attach backend metadata to the request, including: (i) backend type (OpenAPI vs. MCP), (ii)

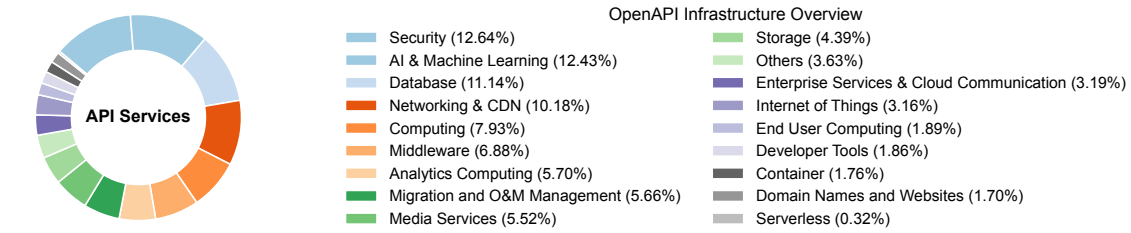


Figure A1. API service overview

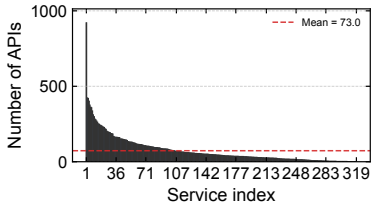


Figure A2. API count distribution

selected transport (HTTP, HTTP+SSE, Streamable HTTP), and (iii) session-routing hints required by the deployment mode (stateless vs. stateful).

- **Rewrite.** Rewrite the tool invocation into an executable backend operation. For an OpenAPI backend, the gateway binds tool arguments to the HTTP method, path, query parameters, headers, and body, including URI template expansion. For an MCP backend, the gateway converts the exposed tool identifier into the backend-visible tool name (e.g., stripping the namespace) and normalizes request fields when MCP variants or versions differ.
- **Forward.** Dispatch the rewritten operation using the annotated transport. For OpenAPI backends, the gateway issues an HTTP request and wraps the response into an MCP tool-call result. For MCP backends, the gateway sends the MCP request over the selected MCP transport and normalizes the response or event stream into an MCP-compliant reply to the host.

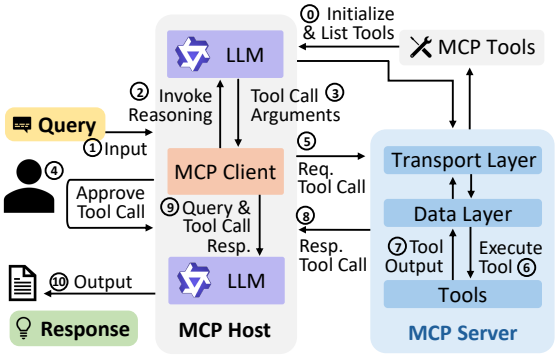


Figure A3. MCP interaction workflow

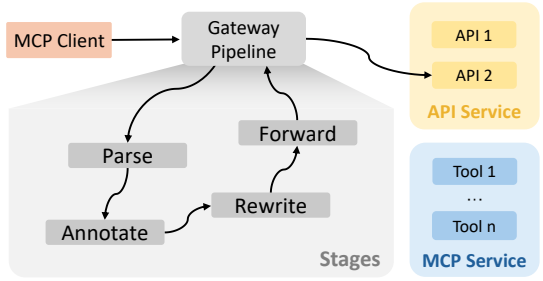


Figure A4. Multi-stage routing decision pipeline

E Choosing Retrieval Candidate Count k

In the retrieve-then-select pipeline, end-to-end accuracy decomposes as:

$$\mathrm{Acc}_{\mathrm{e2e}} = \mathrm{Recall@}k \times \mathrm{Acc}_{\mathrm{select}} \mid \mathrm{recalled} \qquad (1)$$

Retrieval recall increases with k but saturates, while selection accuracy decreases as additional candidates dilute the LLM's attention. Optimizing either term alone is misleading: a small k maximizes selection accuracy but forfeits queries whose ground truth falls outside the candidate set—these queries score zero regardless of LLM capability. For example, at k=5 the LLM achieves 87.6% selection accuracy, yet 13.6% of queries have already lost the correct tool at retrieval, yielding only 75.7% end-to-end. The optimal k balances both terms. We sweep $k \in \{5, 10, 15, 20, 25, 30, 40, 50\}$ and plot both terms together with their product in Figure A5.

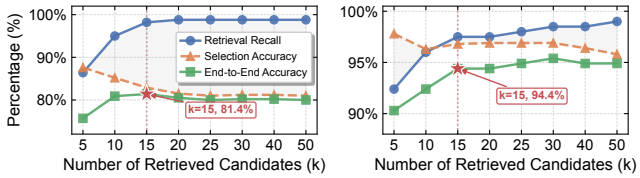


Figure A5. Retrieval recall, LLM selection accuracy, and their product (end-to-end accuracy) as a function of the candidate count k. Left: Alibaba Cloud (N=3,616). Right: ToolBench G1 (N=480).

On the Alibaba Cloud dataset (N=3,616), recall reaches 98.2% at k=15 and gains only 0.6 pp through k=50, while selection accuracy drops from 82.9% to 81.0%. The product peaks at k=15 (81.4%), strictly above all other k values. On ToolBench (N=480), k=15 yields 94.4%, within 1.0 pp of the peak and statistically indistinguishable under a 95% Wilson

confidence interval. We therefore fix k=15 for our default production deployment.

F Handling Long-Lived Connection

Transport modes impose different constraints on session consistency: In Streamable HTTP transport, SSE streams are typically request-scoped and closed after the response is delivered, so long-lived channel ownership by a specific gateway instance is not required. In HTTP+SSE transport, the SSE channel is long-lived and strongly bound: the client maintains a persistent SSE connection to a particular gateway instance, and all server-to-client responses and notifications must traverse that channel. This introduces a structural hazard: a subsequent request carrying the same Session-ID:Frontend may be load-balanced to a different gateway instance that does not own the SSE return channel.

To address this, as shown in Figure A6, the gateway uses a cross-node **Pub/Sub** coordination mechanism and the Session DB mapping between Session-ID:Frontend and Session-ID:Backend. Depending on the frontend/backend transport, the gateway applies:

1. **Frontend Streamable HTTP, backend Streamable HTTP:** the request/response completes on the receiving gateway instance; the gateway consults Session Meta Store as needed to maintain backend session consistency.
2. **Frontend HTTP+SSE, backend HTTP+SSE:** the key requirement is that requests carry Session-ID:Backend so that backend responses are delivered to the correct gateway–backend SSE channel. Any gateway instance may issue the backend request; the backend uses Session-ID:Backend to emit events on the corresponding SSE channel, and the gateway instance holding that channel returns events to the client.
3. **Frontend HTTP+SSE, backend Streamable HTTP:** because the client-facing SSE channel is bound to a specific gateway instance but requests may land elsewhere, the non-owning instance publishes the request (keyed by Session-ID:Frontend) via Pub/Sub. The SSE-owning instance subscribes, executes the request against the backend using Streamable HTTP, and streams the response back to the client over the established SSE channel.
4. **Frontend Streamable HTTP, backend HTTP+SSE:** the HTTP response must return from the gateway instance that received the request, but backend responses arrive at the gateway instance owning the gateway–backend SSE channel. The request-receiving instance initiates the backend call and records a Correlation-ID; the SSE-owning instance publishes received backend events keyed by Correlation-ID; the request-receiving instance subscribes and assembles the Streamable HTTP response.

Overall, the gateway makes the relationship between "request accessing instance" and "response channel ownership" explicit: only when a mismatch could break end-to-end delivery does it use Pub/Sub to transfer execution or relay responses across nodes. This preserves session-centric semantics and availability for long-lived connections while keeping the system elastically scalable.

Table A1. Prerequisite completeness (%) by chain complexity. Baseline and Discovery degrade to 0% at cmplx. ≥5; the Knowledge Cache remains 100% through complexity 12.

		Set-1 (394 tools)			Set-2 (3616 tools)		
Cmplx.	n	**Base.**	**Disc.**	**Cache**	**Base.**	**Disc.**	**Cache**
2	288/362	70.1	93.4	**100**	73.2	92.5	**100**
3	317/327	11.7	84.9	**100**	9.8	85.3	**100**
4	166/213	1.8	43.4	**100**	0.0	30.5	**100**
5	18/52	0.0	0.0	**100**	0.0	0.0	**100**
6	43/62	0.0	0.0	**100**	0.0	0.0	**100**
10	–/168	–	–	–	0.0	0.0	**100**
11	–/21	–	–	–	0.0	0.0	**100**
12	–/6	–	–	–	0.0	0.0	**100**

G Query Skewness Analysis

In production traces, we observe clear query skewness: a small fraction of queries with similar *intents* accounts for the majority of tool-invocation traffic. These semantically similar queries tend to trigger the same (or highly overlapping) set of tools, leading to a long-tailed and hotspot-heavy distribution of tool usage at the intent level. To quantify this effect, we conduct an embedding-based similarity analysis on four production traces. The results are shown in Figure A7.

H Tool Recommendation recall

To validate the efficiency of tool recommendation, we report additional recall results on other production traces (Figure A8). Across traces, tool recommendation achieves consistent, high recall.

I Knowledge Cache: Chain Topology

This appendix provides the detailed breakdown of Knowledge Cache evaluation (Table A1) and illustrates representative dependency chain topologies observed in our evaluation. Each chain consists of API tools connected by foreign-key (FK) parameters, i.e., a downstream tool requires an output parameter produced by an upstream tool. The **Knowledge Cache** precomputes these chains from the API schema and injects the full prerequisite sequence into the retrieval result, achieving 100% prerequisite completeness with zero additional LLM round-trips.

Table A1 breaks down prerequisite completeness by chain complexity, i.e., the number of total related tools, across two tool sets. At complexity 2, both Baseline and Discovery maintain reasonable completeness (70–93%). However, completeness degrades sharply: Baseline drops to near 0% at complexity 4, Discovery follows at complexity ≥5, because

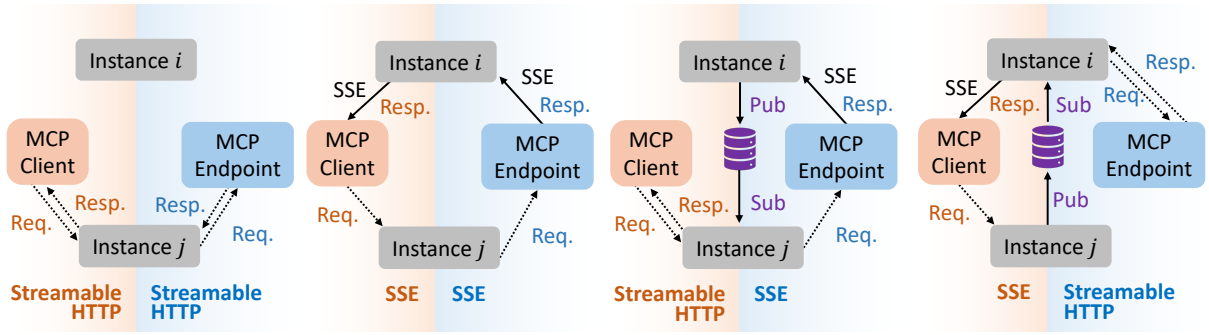
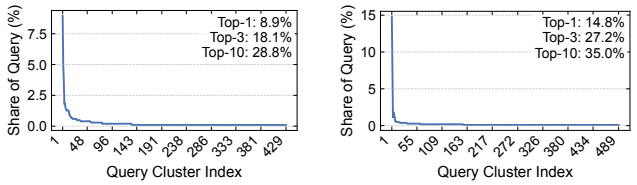


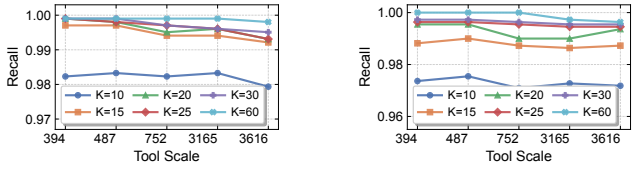
Figure A6. Handling long-lived SSE connections and HTTP response



(a) Skewness of Query Trace 1 **(b)** Skewness of Query Trace 2

(c) Skewness of Query Trace 3 **(d)** Skewness of Query Trace 4

Figure A7. Query skewness in our deployment



(a) Additional Query Trace B **(b)** Additional Query Trace C

Figure A8. Recall v.s. Tool Scale over production traces.

each iterative discovery step has an independent success rate, and the compound probability decays exponentially. The Knowledge Cache remains at 100% through the maximum observed complexity of 12, confirming that static schema-derived chains eliminate the complexity-dependent failure mode entirely.

I.1 Diamond DAG: AssociateRouteTable

The target tool `AssociateRouteTable` requires two FK parameters, `RouteTableId` from `CreateRouteTable` and `VSwitchId` from `CreateVSwitch`, both of which depend on `VpcId` produced by a shared ancestor `CreateVpc`. This diamond pattern requires the agent to independently discover *both* branches and recognize their convergence at the common ancestor. Iterative discovery must issue at least two LLM round-trips to resolve the two branches, and missing either one breaks the entire chain.

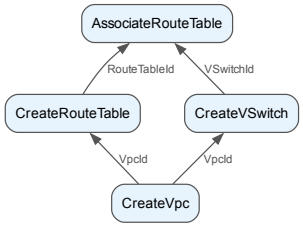


Figure A9. Diamond DAG topology. `AssociateRouteTable` depends on two FK parameters (`RouteTableId`, `VSwitchId`), both of whose producers share a common ancestor `CreateVpc`.

Knowledge Cache

`CreateVpc` → `CreateRouteTable` → `CreateVSwitch` → `AssociateRouteTable`

I.2 Fork-Join DAG: AssociateHaVip

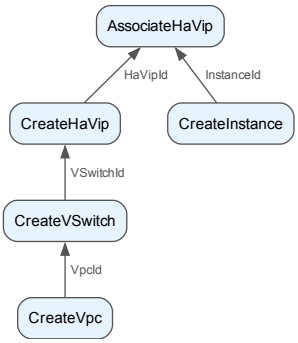


Figure A10. Fork-join DAG topology. `AssociateHaVip` joins a deep branch (depth 4) and a leaf node `CreateInstance` (depth 1).

The target tool `AssociateHaVip` joins two asymmetric branches: a deep chain (`CreateVpc` → `CreateVSwitch` → `CreateHaVip`, depth 4) and a leaf node (`CreateInstance`, depth 1), total complexity 5. The deep branch requires three successive FK resolutions (`VpcId` → `VSwitchId` → `HaVipId`), while the shallow branch provides `InstanceId` directly. This asymmetry means iterative discovery must

handle branches. The deep branch requires multiple LLM round-trips, and the compound probability of resolving all steps drops with each additional hop.

Knowledge Cache

`CreateInstance` → `CreateVpc` → `CreateVSwitch` → `CreateHaVip` → `AssociateHaVip`

I.3 Linear Chain: CreateSslVpnClientCert

Figure A11. Linear chain topology. Each tool depends on exactly one predecessor via a required FK parameter.

A pure linear chain where each tool depends on exactly one predecessor via a required FK parameter: `CreateVpnGateway` requires `VpcId` from `CreateVpc`; `CreateSslVpnServer` requires `VpnGatewayId` from `CreateVpnGateway`; and `CreateSslVpnClientCert` requires `SslVpnServerId` from `CreateSslVpnServer`. Each discovery step has an independent success rate, explaining why iterative discovery achieves 30.5% completeness at complexity 4 on Set-2, dropping to 0% at complexity ≥ 5.

Knowledge Cache

`CreateVpc` → `CreateVpnGateway` → `CreateSslVpnServer` → `CreateSslVpnClientCert`

I.4 Complex Chain: DeleteAutoProvisioningGroup

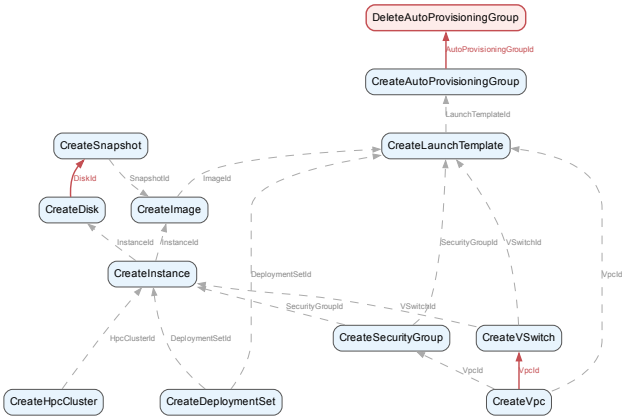


Figure A12. Complexity 12 dependency chain, spanning three cloud service domains. Solid red arrows denote hard (required) FK prerequisites; dashed gray arrows denote soft (optional) FK prerequisites. The target node is highlighted in red.

The complex dependency chain in our evaluation spans 12 tools across three cloud service domains (VPC networking, ECS compute, and ECS storage). The target `DeleteAutoProvisioningGroup` sits at the end of a complex DAG with 3 hard prerequisites (solid arrows) and 14 soft prerequisites (dashed arrows). The critical path traverses: network provisioning (`CreateVpc` → `CreateVSwitch`), compute instance creation (`CreateInstance`), storage lifecycle (`CreateDisk` → `CreateSnapshot` → `CreateImage`), launch template composition (`CreateLaunchTemplate`), and auto-provisioning group management (`CreateAutoProvisioningGroup` → `DeleteAutoProvisioningGroup`). Three leaf nodes (`CreateDeploymentSet`, `CreateHpcCluster`, `CreateVpc`) serve as independent entry points.

This chain is representative of real-world cloud infrastructure workflows where a single user intent, deleting an auto-provisioning group, implicitly requires orchestrating resources across multiple service boundaries. As the agent would need to traverse cross-domain FK dependencies spanning compute, networking, and storage APIs, knowledge that is unlikely to be captured by LLM. The Knowledge Cache resolves the full chain in a single retrieval step with zero additional LLM round-trips.

Knowledge Cache

`CreateDeploymentSet` → `CreateHpcCluster` → `CreateVpc` → `CreateSecurityGroup` → `CreateVSwitch` → `CreateInstance` → `CreateDisk` → `CreateSnapshot` → `CreateImage` → `CreateLaunchTemplate` → `CreateAutoProvisioningGroup` → `DeleteAutoProvisioningGroup`