Scaling LLM Agent Tool Access: The MCP Gateway

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 complete tasks. Model Context Protocol (MCP) standardizes tool calling between LLM agents and tool providers. It reduces integration cost and drives rapid growth in tool scale. In production deployments, tool access becomes difficult to scale. On the tool provider side, legacy services are not directly callable through MCP; the rapid protocol development creates ongoing compatibility cost. On the agent side, the number of mounted tools 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 MCP Gateway system. 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 at the gateway. The conversion module supports 315 OpenAPI services in the cloud and integrates MCP transport variants. Session-aware routing decouples the scalability of the gateway layer and the MCP service layer. The hybrid retrieval engine sustains 98% Top-15 recall; it scales agent tool access to 3,000+ tools with improved tool selection accuracy. We also share the lessons learned from deploying the gateway system in production.

CCS Concepts: • **Networks** → **Cloud computing**; **Application layer protocols**; **Middle boxes / network appliances**.

Keywords: Model Context Protocol, Gateway, Cloud Networks

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



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
SIGOPS ATC 2026, Hong Kong, China
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857007

ACM Reference Format:
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, and Shunmin Zhu. 2026. Scaling LLM Agent Tool Access: The MCP Gateway. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15-18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3828523.3857007

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 [22, 33–35, 49, 60, 61, 64, 68]. 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) [18]. 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 [38, 40]). Also, MCP traffic in the cloud is growing rapidly (Figure 2).

However, building an agent that can access massive tools with reliable tool invocation faces several challenges. (1) The number of public APIs has reached 10^6 [55], yet these existing web era APIs are not directly callable by agents through MCP. (2) Tool developers must handle work orthogonal to tool functionality, such as MCP compatibility issues and the integration of authentication. (3) A single agent can only mount a limited number of tools, insufficient for diverse tasks at scale. Mounting too many tools may exceed the context window of the LLM, increasing token cost and latency while reducing task success rate. (4) MCP assumes a client–server direct-connect model. For availability and load balancing, servers are often deployed with multiple replicas. The Host must ensure *session-aware routing* for stateful servers (i.e., all requests in the same logical session are forwarded to the replica that maintains the context).



Figure 1. MCP

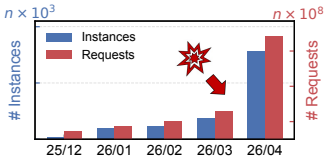


Figure 2. Traffic in one cloud region

To address these issues, we introduce an MCP Gateway, breaking the direct-connect model. While MCP runs over HTTP, its key routing information resides in the HTTP body rather than headers, preventing existing web-oriented application load balancers (e.g., L7 load balancers and API Gateways) [8–11] from being directly reused without non-trivial customization. We implement the following capabilities in the MCP Gateway. (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 adaptation cost. (2) *Function Offloading.* To support fine-grained tool access control at scale, we implement authentication modules that enable identity-to-tool mappings. (3) *Tool Recommendation.* We design retrieval methods for MCP. Beyond naive matching, we exploit the fact that MCP tool schemas 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. To address the drop in recommendation accuracy when generic matching fails to capture domain-specific knowledge, we further design a query-rewrite method and a knowledge cache based on domain knowledge extraction. (4) *Session-aware Routing.* The gateway offloads request distribution from the host. 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 the deployed cloud-scale MCP Gateway system for massive tool access.** Protocol adaptation enables direct access to legacy APIs with low conversion overhead (P50 $\leq$ 143 μs at 100 fields). The tool recommendation module achieves 98% recall for Top-15 tools when users mount $O(1k)$ tools, with retrieval latency below 250 ms. This reduces time cost and improves agent tool selection accuracy with Top-15 tool recommendations.
- **We address stateful request routing in multi-gateway deployments.** A centralized store-based session synchronization is proposed to support session-aware routing. We further introduce a Pub/Sub mechanism to synchronize responses across gateway instances to preserve long-lived connections for stateful backends. We show the overhead and explore the trade-offs of session management.
- **We share production deployment lessons.** (1) Maintaining long-lived gateway–backend connections reduces mean latency by up to 51.7%. (2) Leveraging tool dependencies and locality, knowledge caching returns complete prerequisite tool chains, and LRU semantic caching achieves up to a 2.29× speedup in average recommendation latency per workload. (3) We further discuss an encoding-based session synchronization for cross-gateway coordination, and (4) propose a rule-extraction method, improving tool recommendation Top-15 recall 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 [27, 34, 37, 73]. Unlike early chatbots that focused on text generation without external interaction, LLM agents can plan, use tools, and accomplish complex tasks. Using reasoning-and-acting paradigms, e.g., ReAct, CoT, and ToT, agents decompose user intents into multi-step tasks [65, 71, 72] and interact with external systems, e.g., querying databases, managing cloud resources, and triggering workflows [22, 33, 34, 49, 60, 61, 64, 68]. The LLM acts as a controller that selects tools, emits structured arguments, and triggers execution (i.e., tool calls) [46, 54, 67]. As a result, LLM agents require robust tool infrastructure to mediate agent–tool interactions [14]. The absence of widely adopted protocols remains a major bottleneck to interoperating with external tools and scaling agent ecosystems, motivating recent standardization efforts [70]. The MCP [43] addresses this challenge by defining a standard interface between agents (*MCP Hosts*) and external resources (*MCP Servers*), facilitating tool integration.

2.2 The Rise of Model Context Protocol

Interoperability via interface standardization. Before the MCP, LLM applications and agent frameworks [16, 24, 25] typically integrated external tools through bespoke adapters, resulting in a fragmented ecosystem [30]. Heterogeneous tool interfaces and provider-specific calling schemas forced point-to-point integration, so connecting M LLM applications to N tools conceptually approached an $O(M \times N)$ engineering burden. MCP reduces this complexity to $O(M+N)$ by standardizing 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 [43]. 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 host invokes *List Tools* to fetch tool definitions from one or

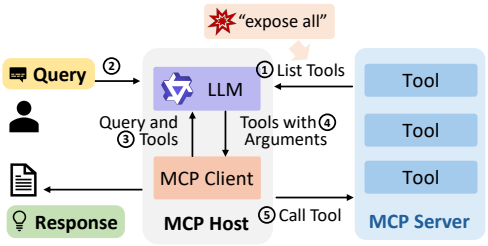


Figure 3. A concise MCP interaction workflow

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 task completed and returns the final response.

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

Legacy infrastructures. Legacy cloud platforms are primarily built around OpenAPI/Swagger endpoints, engineered for developers and conventional SDKs 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 [56], whereas MCP has existed for only about two years since its public release [5]. The MCP community registry currently contains on the order of 10^4 MCP servers [38, 40]. 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 [55]. Alibaba Cloud exposes on the order of 10^4–10^5 OpenAPI operations [4, 62, 63], suggesting that the 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.

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 [26]	**1M**	774
Qwen-Plus [58]	**1M**	627
GPT-5.2 [48]	**400k**	270
Qwen3-MAX [59]	**256k**	198
Claude-Opus-4.5 [6][7]	**200k**	125
DeepSeek-V3.2 [21]	**128k**	98

Table 2. For the VPC agent with ~60 tools, a duplicating 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 [6][7]	92,924 × 10	$0.999
Qwen3-MAX [59]	80,085 × 10	$0.685
GPT-5.2 [48]	27,100 × 10	$0.090
Gemini-3-Flash [26]	65,016 × 10	$0.062
DeepSeek-V3.2 [21]	77,564 × 10	$0.041
Qwen-Plus [58]	79,947 × 10	$0.090

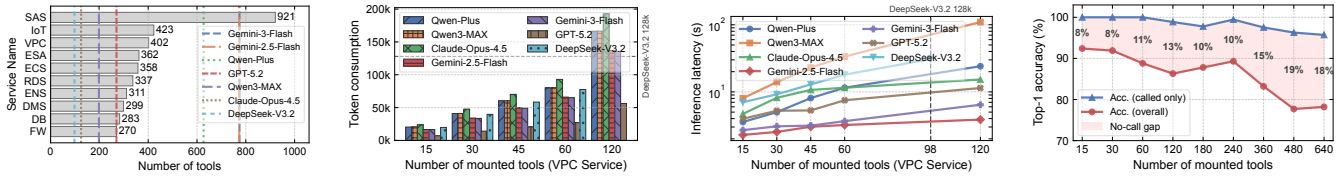
Challenge 1: *Large-scale legacy OpenAPI infrastructures are not MCP-native, making most production APIs inaccessible to agent tool use.*

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 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 [47] show that only a small fraction of servers support both transports, while the rest support only one. Client support also lags behind: in the public market, over 50% of MCP clients support only HTTP+SSE and lack effective support for Streamable HTTP [28]. Second, authentication divergence. Early MCP specifications did not define a standard authentication mechanism [18], leading servers to adopt heterogeneous, vendor-specific security methods. As authentication requirements evolve, deployments must absorb this heterogeneous authentication ecosystem to preserve compatibility, rather than requiring server developers to repeatedly re-implement authentication logic.

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

[1]Measured in our experiments (January 2026) using a tool set derived from 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.

generate 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 history, 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 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 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 Qwen models, or more than 56k tokens even with efficient tokenization (Figure 4b). This verbosity increases inference cost (Table 2) and crowds out agent tasks. **(3) Prohibitive inference latency.** Prompt size grows with the tools, slowing tool selection and argument generation. Figure 4c shows that for Qwen3-MAX, mounting 120 tools already pushes latency (prompting + tool selection + argument generation) beyond 100s, while 180 tools exceed 200s. Such delays break interactivity. **(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%), degrading agent performance. These failures are analogous to the well-known "lost-in-the-middle" phenomenon [13, 36].

> ***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 access to large catalogs without exhausting 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 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 the **MCP Gateway System**. The gateway breaks

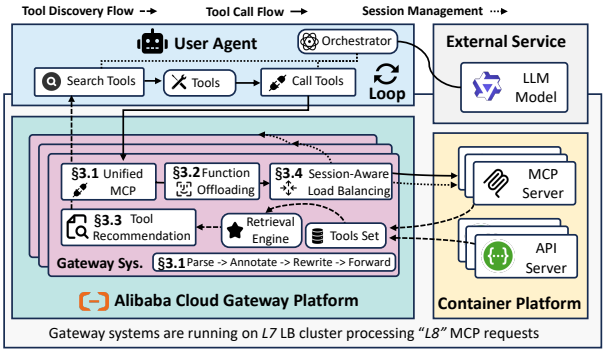


Figure 5. The MCP Gateway architecture

the original direct-connect model by interposing a shared control point between MCP clients and tool servers.

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 MCP Gateway provides these shared functions as common infrastructure and enforces them on the end-to-end request path. Via 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 search iteratively in a loop, with or w/o LLM reasoning. **(3) Deterministic and efficient execution.** Instead of relying on an 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, which consistent hashing [31] alone cannot provide. This requires consistent routing state to be shared 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.

4 Implementation

We implement the MCP 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 (~6K LoC) loaded by the gateway runtime, 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 enable 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.

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 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. 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.

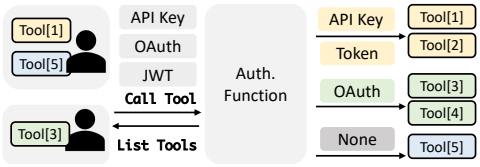


Figure 6. Fine-grained access-control function offloading

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. Each client and backend uses 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. The gateway enforces identity-based, tool-level authorization consistently across heterogeneous backends by centralizing authentication management. Specifically, it grants the access of *List Tools*, *Call Tool*, and the meta tool *Tool Search* (discussed in §4.3) to each authenticated user, ensuring that each authenticated user can only discover and invoke tools permitted by policy.

4.3 Hybrid Tool Recommendation

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

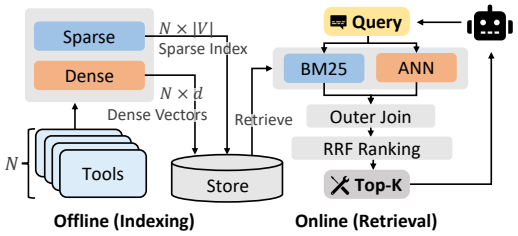


Figure 7. Hybrid tool retrieval engine

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 returned 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.

Formally, each retrieval path returns L candidates (L=120). A tool t in the union is scored by

$$\text{score}(t) = \sum_{i \in \{\text{dense, sparse}\}} \frac{\mathbb{1}[t \in C_i]}{\text{RRF}_k + \text{rank}_i(t)} \qquad (1)$$

where $\text{rank}_i(t)$ is the rank of t in C_i, RRF_k=20 is the smoothing constant. The dense path uses cosine similarity; the sparse path uses BM25 (k_1=1.2, b=0.75). Both the dense and sparse indexes are constructed from tool names, descriptions, and top-level parameter names; the sparse path also splits camelCase identifiers and removes stop words.

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

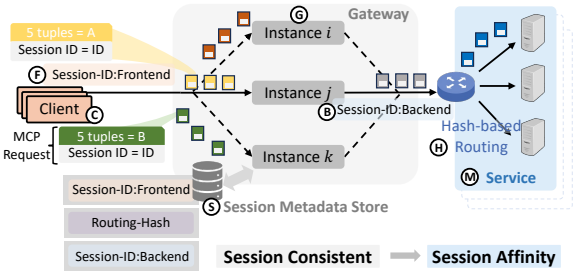


Figure 8. Session-aware request routing

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 MCP Gateway (Ⓖ) terminates client (Ⓒ) connections and establishes backend connections to MCP Services (Ⓜ) on demand. 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 does not track instance churn inside each service. Since the frontend session is established before the backend session, the gateway computes a `Routing-Hash` after generating `Session-ID:Frontend`, and places it into an HTTP request header on the *Gateway -> Service* path as the key for intra-service 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. `Routing-Hash` is stored as part of the session metadata, together with `Session-ID:Frontend` and its corresponding `Session-ID:Backend`. 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. Each MCP Service exposes a single domain endpoint and applies stateless consistent hashing [31] to the

Routing-Hash (Ⓗ) inside the service, mapping requests stably to a specific backend MCP server. Only a small fraction of sessions should be re-established by the gateway when the backend service scales or fails.

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 mismatch in a multi-instance gateway, as 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, 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–gateway SSE channel; when the backend uses SSE, responses are 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.

5 Deployment

Production Scale. MCP Gateway has operated continuously in production for more than 11 months across more than five Alibaba Cloud regions. Over this period, tens of thousands of gateways have been deployed. Each gateway runs as multiple instances and scales out as its workload grows, so the total number of instances is several times the number of gateways. Session state is synchronized only among the instances of the same gateway; the overhead thus depends on a gateway's instance count, not on the total deployment size. The OpenAPI conversion module supports OpenAPI 3.0 and 3.1 and has been validated on more than 20,000 OpenAPI operations across 315 Alibaba Cloud products. The largest observed per-user mount reaches thousands of tools.

6 Evaluation

6.1 Evaluation Setup

Environment. 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. We evaluate in two settings. Production evaluation demonstrates scalability in one of our largest clusters and tool retrieval on production tool sets. Controlled evaluation runs in a separate cluster for concurrent session traffic stress testing and for baseline comparison by replaying production workloads, since online traffic lacks ground truth and the baselines cannot serve real tenants.

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* [51] (up to 640 tools): a public benchmark for comparability.

Baselines. (1) *RAG-MCP* [23]: dense embeddings only; (2) *ToolLLM* [51]: retriever fine-tuned on ToolBench; (3) *BM25*: sparse retrieval; (4) *Expose-all*: 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).

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
Expose-all	55662.68	55029.86	59962.32	67936.70

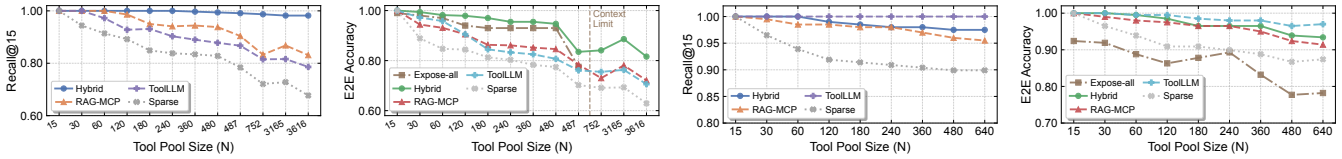
Table 4. Token consumption, Qwen3-Plus

	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
Expose-all	505.86	507.29	511.38	512.56

6.2 Agent Tool Selection Performance

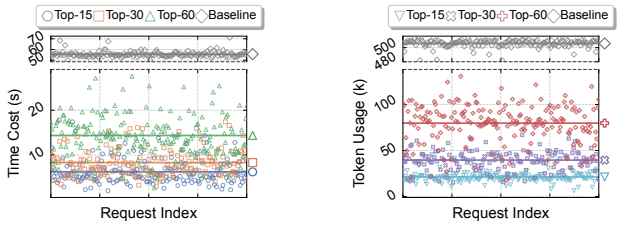
Tool selection accuracy on cloud-scale tools. Any retrieval and selection pipeline cuts tokens and latency; we first investigate whether it could preserve or improve the accuracy over the expose-all approach. As shown in Figure 9b and 9d, the accuracy of the expose-all approach degrades as the tool pool grows: on ToolBench, the selection accuracy drops from 92.4% at N=15 to 78.2% at N=640, which is worse than all retrieval-based methods; on Alibaba Cloud tools, it exceeds the LLM context window at $N \geq 752$ and becomes infeasible for benchmarking. In contrast, retrieval-based methods maintain feasible accuracy, at N=3,616, Hybrid achieves 81.6% versus 72.1% for RAG-MCP and 70.6% for ToolLLM. Thus, the hybrid retrieval we applied achieves scalability and benefits most for accuracy.

Tool selection latency and token cost. Figure 10 shows that the end-to-end agent tool use completion evaluation follows the Agent–MCP Gateway–Tools path and uses representative cloud tools and production user traffic (VPC, ECS and SLB tools) as a reference point. The gateway hosts $O(1k)$ tools for cloud resource operations. In the expose-all setting, the agent selects the target from 372 tools, since it cannot scale to $O(1k)$ tools. In the retrieval-based setting, the agent



(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

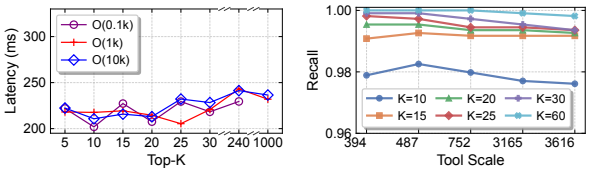


Figure 11. Retrieval latency v.s. Tool scale

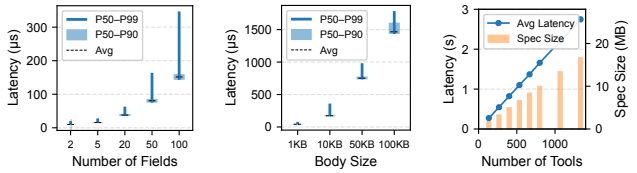
Figure 12. Top-K recall v.s. Tool scale

selects tools from the recommended list (Top-15, 30, 60), and then issues tool calls through the MCP Gateway. As shown in Table 3, on average, Top-15 reduces time cost from 55.7s (expose-all) to 6.2s, 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.[2] The agent can benefit this only if the retrieval maintains high recall.

6.3 Hybrid Tool Recommendation

Latency and recall at scale. Figure 11 shows that latency remains stable from $O(0.1k)$ to $O(10k)$ tools. Across different retrieval result set sizes, i.e., Top-5, Top-10, Top-15, and Top-1000, the retrieval process complete within 202–243 ms, which is bounded. As illustrated in Figure 12, recall also remains high as the tool set grows. Note that, as the mounted tool set increases from 394 to 3616 tools, Top-15 retrieval can still achieve over 98% recall. It further shows a recall gain when expanding the recommendation set from Top-10 to Top-15. This indicates Top-15 is an effective result set size for tool recommendation. Beyond these, our production traces reveal additional types of *weak-semantic* queries (e.g., ID-only requests). The semantic signal of these requests is too

[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).

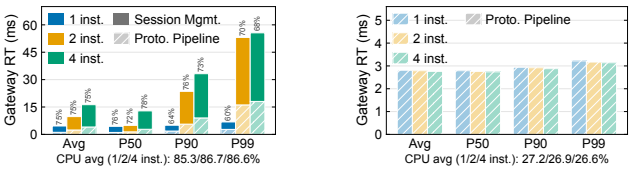
weak for retrieval. We report this observation and production optimization as (Lesson 4) in §7 .

Comparison with retrieval baselines. On ToolBench (Figure 9c and 9d), ToolLLM achieves the best recall across all sizes, since ToolLLM's retriever was trained (fine-tuned) over ToolBench datasets. However, this advantage does not transfer on other domain, if we do not re-tune it. On Alibaba Cloud tools (Figure 9a and 9b), ToolLLM's recall drops as the tool pool grows, falling to 78.6% at N=3,616, below its performance on ToolBench. In production, the tool set may change according to tenants continuously, such re-tuning may become impractical, because every update requires data generation, retraining, and redeployment.

Without fine-tuning, the hybrid retrieval consistently outperforms 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 good at large scale. For the cloud tools, dense embeddings capture intent-level information, while BM25 preserves exact matches on resource identifiers and parameter names. 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).

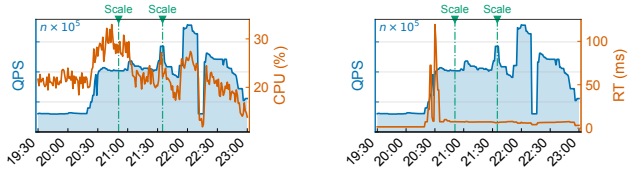
6.4 Gateway Overhead and Scalability

API conversion overhead is negligible. For the online part, 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, the median conversion latency stays below 35 μs. For the largest payload with 100 fields, the P50 latency is 143 μs. Figure 13b shows that the conversion scales linearly with body size. Figure 13c reports the offline



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

Figure 14. Gateway RT breakdown vs. # instances



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

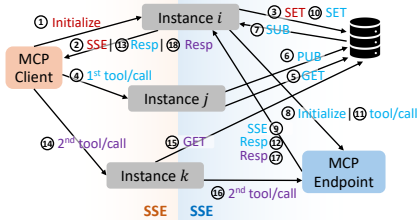
Figure 15. The scalability of the MCP Gateway

conversion that generates MCP tools from API specs. For a large one of 16.7 MB, generating 1,334 tools takes only 2.75 s. The conversion time grows linearly with the number of tools, it runs once per registration, the cost can be amortized.

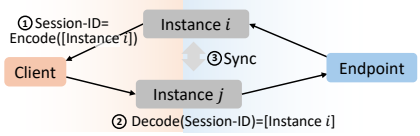
Session-aware routing dominates per-request cost. Figure 14a shows that stateful routing adds substantial overhead, even before horizontal scale-out. With a single gateway instance at ~100 QPS (under stress with concurrent sessions), the mean RT is 4.59 ms with P50/P90/P99 of 4.41/5.02/6.81 ms. Session 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 requests of a session are distributed to an instance that does not hold the session, the gateway must fetch session metadata from the store. With two instances, P50 latency remains low as the single-instance while P90 increases to 23.58ms, with four instances, the latency rises further and makes P50 to 12.87ms. We can compare this with stateless ones in (Figure 14b), 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).

Elastic scale-out with stable RT. Figure 15 shows a production trace from one of our largest clusters. It 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, after traffic rises around 20:30 and per-instance



(a) Centralized architecture



(b) Session-ID encoding architecture

Figure 16. From centralized management to encoding.

CPU increases from ~20% to ~30% (Figure 15a), the cluster scales and drives CPU back toward the ~20% steady level without sacrificing RT. A second scale-out occurs at 21:35:00 under a QPS-driven CPU increase, again with stable RT. 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.

Each instance caches session metadata locally. It accesses the centralized store only when (i) a new session first lands on it, or (ii) a newly added instance first sees an existing session. Figure 14 stresses the gateway with concurrent new-session traffic. Figure 15 shows that in production, a session carries more than one request, so the cost of the first miss is amortized over subsequent requests. Scale-out adds only a few instances as needed (typically 1–4). As a result, RT stays flat across scale-out events.

7 Lessons Learned from Deployments

Lesson 1: Coordination for session consistency. In production, we observed that when both frontend and backend sessions use SSE, a session is initialized on one gateway instance while subsequent tool calls can be 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). 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 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 instances, but it is complex and can generate substantial synchronization traffic, especially under elastic scaling.

However, there is a property of MCP traffic, the gateway-assigned `Session-ID` should be carried by the MCP client on every subsequent request. This makes the `Session-ID` an effective signal for routing and coordination. Our system is evolving toward a *Session-ID encoding* approach (not

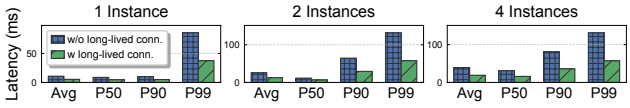


Figure 17. Latency w/o and with long-lived connection

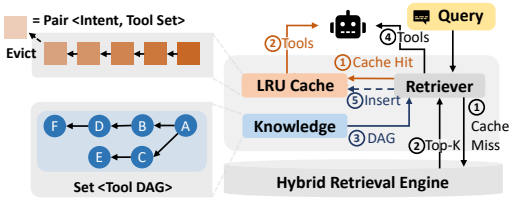


Figure 18. Semantic cache

yet deployed at scale). As shown in Figure 16b, during session establishment, the gateway encodes an owner hint (e.g., an instance identity) 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 session and connection termination. Deployment measurements show that this design introduces substantial user-aware 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 100ms. The lesson is that, when the backend requires long-lived connections and sessions, the gateway should absorb this complexity to avoid overhead on the critical path. After the redesign, long-lived gateway–backend connections reduce mean/P90/P99 latency by up to 51.7%/55.7%/56.5%.

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

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

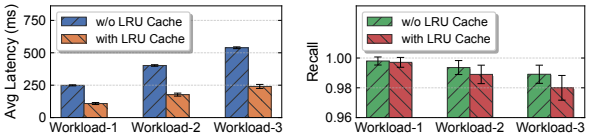


Figure 19. LRU caching on latency with varied production trace workloads (a); and recall with different query sets (b)

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**

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 below or around 90%, so the probability of recovering the full chain decays exponentially with depth.

(1) Knowledge Cache. To eliminate the above failure, the MCP Gateway introduces a *Knowledge Cache* that precomputes prerequisite chains from the tool schemas (Figure 18). At indexing time, the gateway analyzes 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 combines the prerequisite tools into the final result, so the agent receives an executable chain in one shot, with 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 on average. Also, the success rate can drop to 0% completeness at chain depth ≥5 in our production measurements.

(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 production trace into 224 clusters, and the top 10 clusters account for 80.2% of the traffic. The MCP Gateway adds an LRU *semantic cache* for ⟨*Intent*, *Tool Set*⟩ pairs to bypass external retrieval on cache hits. Even a small cache (e.g., 64 entries) speeds up retrieval by up to 2.29× under varied workloads with negligible recall loss (only related to the semantic characteristics of query workloads) (Figure 19a, Figure 19b).

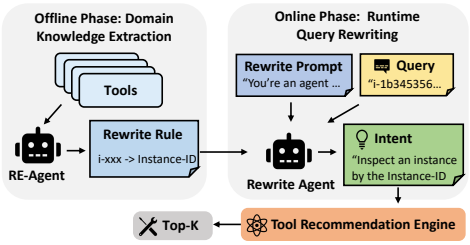


Figure 20. Rewrite-Agent and RE-Agent workflow

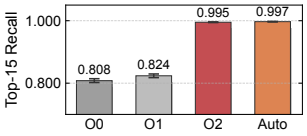


Figure 21. Query rewrite optimizations (on Trace-3)

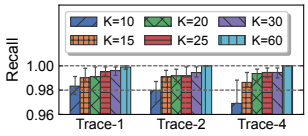


Figure 22. Top-K recall v.s. Traces

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, under-specified intents. In practice, many user queries are weak at semantic. 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 semantic 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) to 82.4% (O1), 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 recommends the Top-K tools based on the intent (O1 in Figure 21). The key is *entity normalization*, mapping identifiers and other non-semantic strings 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 recall, Top-10 recall exceeds 96%, Top-15 and larger recommendation sets exceed 98%.

The offline rule extraction cost is amortized over the Q queries between two tool-set changes; the per-query cost of the pipeline is

$$C_{\text{RE-Agent}} = \frac{C_{\text{extract}}(\Delta N) + C_{\text{deploy}}}{Q} + C_{\text{rewrite}} + C_{\text{retrieval}} \quad (2)$$

where C_{extract} grows only with the changed tools ΔN. In our measurements, a full pass over 3,616 tools takes ~60 s and yields ~200 rules. The online rewriting adds a median of ~560ms. The cost is not very high. This is because a small ~3B activated MoE model suffices, and the KV cache of shared rules can be reused across queries. This cost avoids extra LLM tool selection round-trips (Table 3), which are more expensive. Currently, the rewriting is enabled per tenant if their workflows contain ID-only/weak-semantic queries; otherwise queries use plain hybrid retrieval.

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

8 Discussion

Cloud gateway design choices. We implemented the protocol conversion using WebAssembly (WASM). This isolates the pipeline from the host, ensuring that each stage remains modular for fast-evolving requirements. We deployed the MCP integration in the cloud rather than alongside the client, so credentials can be managed securely instead of being distributed to clients. Also, services and the retrieval engine scale independently without the agent being aware of it.

OpenAPI conversion limitations. Currently, pagination and polling can be handled by the agent via multiple calls, but callbacks initiated by a backend cannot be received by the gateway. We plan to implement an event-bridging layer on the gateway to address this limitation.

Future work for security bridging. Bridging ingress and egress authentication makes the gateway a trust boundary that holds backend credentials on behalf of agents. Current

Table 6. Comparison with open-source and commercial gateway systems (October 2026). ● supported, ◑ partial, ○ absent or undocumented.

	API	Transport	Auth	Search	Session
MCP Gateway	●	●	●	●	●
AgentCore [2]	●	○	●	◑	◑
Microsoft [42]	○	○	◑	○	◑
agentgateway [3]	●	●	●	○	◑
LiteLLM [17]	●	◑	●	◑	◑
ToolHive [20]	○	◑	●	●	◑

mitigations include fine-grained access control. Our next steps involve isolating credential storage by decoupling traffic forwarding from credential management and supporting credential rotation to secure the system.

9 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 [45, 53, 66]. There are also open-source L7 gateways and load balancers (e.g., NGINX and Envoy) [8–11]. 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 parsing JSON-RPC payloads required by MCP, and they cannot directly differentiate tool-invocation requests or have 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 [32, 50, 73] and scheduling inference engines for agentic programs [1, 12, 15, 34, 37, 57]. Orthogonal to our focus on scalable tool access for agents, Cortex [52] reduces latency and cost for remote knowledge/data access in LLM agents via semantic-aware caching, whereas our MCP Gateway 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 [44, 69, 74], reasoning frameworks [65, 71, 72], and agent runtimes [41], to improve task performance and robustness. The MCP Gateway targets the tool-calling infrastructure, providing efficient and scalable tool access. Other studies improve tool selection and argument generation [46, 51, 54] and enhance tool-calling reliability under large tool sets [23]. Recent MCP-related studies examine protocol adoption, reliability, and security [29].

MCP Integration gap analysis. As shown in Table 6, parallel to the development and deployment of our MCP Gateway, open-source and commercial systems [2, 3, 17, 19, 20, 39, 42] have also adopted MCP integration to standardize agent tool access. Our work is beyond merely handling the MCP protocol layer; its value lies in its design and measurements tailored to actual MCP workloads and scenarios.

In terms of capabilities, API conversion and authentication are now common; the key difference lies in the granularity. The first gap is the transport protocol conversion, particularly enabling interaction between HTTP+SSE and Streamable HTTP. agentgateway [3] and our system achieve this. LiteLLM [17] supports this by reconstructing a session for every call, which introduces overhead according to our measurements. AgentCore [2] and the Microsoft MCP Gateway [42] support single or latest transport. The second is session-aware routing. To the best of our knowledge, MCP Gateway is the first work to present the design and measurements of session affinity and session consistency while decoupling scaling of the gateway layer and the MCP service layer. Furthermore, our system handles SSE bridging across multiple gateway instances and backends as a part of session-aware routing. The third is the retrieval engine. ToolHive [20] and our system offer hybrid retrieval. Our advantage comes from production workloads and the corresponding components, e.g., RE-Agent and Knowledge Cache.

10 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 MCP 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 MCP Gateway enables massive tool access and scales agent tool access in production.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful feedback. This work was supported by the National Key R&D Program of China under Grant 2023YFB4502400; the National Natural Science Foundation of China under Grants U24B20153, 62402212, 62272223, and U22A2031; the Natural Science Foundation for Young Scientists of Jiangsu Province under Grant BK20241245; and the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China under Grant JYB2025XDXM901.

References

[1] Saurabh Agarwal, Marco Laju, Donghyun Son, Nitin Kedia, Myungjin Lee, Jayanth Srinivasa, and Aditya Akella. 2026. Nalar: Workflow-Aware Management of Agentic Applications. arXiv:2601.05109 [cs.DC] https://arxiv.org/abs/2601.05109

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

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

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

[5] Anthropic. 2024. *Introducing the Model Context Protocol.* https://www.anthropic.com/news/model-context-protocol Accessed: 2026-01-06.

[6] Anthropic. 2025. *Claude Opus 4.5.* https://platform.claude.com/docs/en/models/opus-4-5/overview Accessed: 2026-10-07.

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

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

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

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

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

[12] Zhuohang Bian, Feiyang Wu, Zhuoran Li, Teng Ma, and Youwei Zhuo. 2026. TokenCake: A KV-Cache-centric Serving Framework for LLM-based Multi-Agent Applications. arXiv:2510.18586 [cs.DC] https://arxiv.org/abs/2510.18586

[13] Thierry Blankenstein, Jialin Yu, Zixuan Li, Vassilis Plachouras, Sunando Sengupta, Philip Torr, Yarin Gal, Alasdair Paren, and Adel Bibi. 2026. BiasBusters: Uncovering and Mitigating Tool Selection Bias in Large Language Models. In *International Conference on Learning Representations*, C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (Eds.), Vol. 2026. 102460–102489. https://proceedings.iclr.cc/paper_files/paper/2026/file/a79875cc0d046ce7ce65f03f3affaa9e-Paper-Conference.pdf

[14] Alan Chan, Kevin Wei, Sihao Huang, Nitarshan Rajkumar, Elija Perrier, Seth Lazar, Gillian K. Hadfield, and Markus Anderljung. 2025. Infrastructure for AI Agents. arXiv:2501.10114 [cs.AI] https://arxiv.org/abs/2501.10114

[15] 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:2508.06948 [cs.DC] https://arxiv.org/abs/2508.06948

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

[17] LiteLLM Contributors. 2026. *LiteLLM AI Gateway.* https://github.com/BerriAI/litellm Accessed: 2026-09-29.

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

[19] MCP Hub Contributors. 2026. *MCP Hub.* https://github.com/samanhappy/mcphub Accessed: 2026-01-06.

[20] ToolHive Contributors. 2026. *ToolHive.* https://github.com/stacklok/toolhive Accessed: 2026-09-29.

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

[22] Meihao Fan, Ju Fan, Nan Tang, Lei Cao, Guoliang Li, and Xiaoyong Du. 2025. AutoPrep: Natural Language Question-Aware Data Preparation with a Multi-Agent Framework. *Proc. VLDB Endow.* 18, 10 (June 2025), 3504–3517. doi:10.14778/3748191.3748211

[23] Tiantian Gan and Qiyao Sun. 2025. RAG-MCP: Mitigating Prompt Bloat in LLM Tool Selection via Retrieval-Augmented Generation. arXiv:2505.03275 [cs.AI] https://arxiv.org/abs/2505.03275

[24] Dawei Gao, Zitao Li, Xuchen Pan, Weirui Kuang, Zhijian Ma, Bingchen Qian, Fei Wei, Wenhao Zhang, Yuexiang Xie, Daoyuan Chen, Liuyi Yao, Hongyi Peng, Zeyu Zhang, Lin Zhu, Chen Cheng, Hongzhu Shi, Yaliang Li, Bolin Ding, and Jingren Zhou. 2024. AgentScope: A Flexible yet Robust Multi-Agent Platform. arXiv:2402.14034 [cs.MA] https://arxiv.org/abs/2402.14034

[25] Dawei Gao, Zitao Li, Yuexiang Xie, Weirui Kuang, Liuyi Yao, Bingchen Qian, Zhijian Ma, Yue Cui, Haohao Luo, Shen Li, Lu Yi, Yi Yu, Shiqi He, Zhiling Luo, Wenmeng Zhou, Zhicheng Zhang, Xuguang He, Ziqian Chen, Weikai Liao, Farruh Isakulovich Kushnazarov, Yaliang Li, Bolin Ding, and Jingren Zhou. 2025. AgentScope 1.0: A Developer-Centric Framework for Building Agentic Applications. arXiv:2508.16279 [cs.AI] https://arxiv.org/abs/2508.16279

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

[27] 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* (Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 415–430. doi:10.1145/3731569.3764814

[28] 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:2509.25292 [cs.CY] https://arxiv.org/abs/2509.25292

[29] Yongjian Guo, Puzhuo Liu, Wanlun Ma, Zehang Deng, Xiaogang Zhu, Peng Di, Xi Xiao, and Sheng Wen. 2026. MCPXkit: The Unified Toolkit for Analyzing Model Context Protocol Security. *IEEE Transactions on Dependable and Secure Computing* 23, 5 (2026), 9796–9811. doi:10.1109/TDSC.2026.3695553

[30] Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. 2026. Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions. *ACM Trans. Softw. Eng. Methodol.* 35, 10, Article 302 (Sept. 2026), 37 pages. doi:10.1145/3796519

[31] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing* (El Paso, Texas, USA) *(STOC '97)*. Association for Computing Machinery, New York, NY, USA, 654–663. doi:10.1145/258533.258660

[32] 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* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[33] Guoliang Li, Xuanhe Zhou, and Xinyang Zhao. 2024. LLM for Data Management. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4213–4216. doi:10.14778/3685800.3685838

[34] 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)*. USENIX Association, Santa Clara, CA, 929–945. https://www.usenix.org/conference/osdi24/presentation/lin-chaofan

[35] Samuel Lin, Jiawei Zhou, and Minlan Yu. 2025. An LLM-based Agentic Framework for Accessible NetworkControl. *SIGMETRICS Perform. Eval. Rev.* 53, 2 (Aug. 2025), 15–20. doi:10.1145/3764944.3764949

[36] 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. doi:10.1162/tacl_a_00638

[37] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. 2025. Autellix: An Efficient Serving Engine for LLM Agents as General Programs. arXiv:2502.13965 [cs.LG] https://arxiv.org/abs/2502.13965

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

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

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

[41] Kai Mei, Xi Zhu, Wujiang Xu, Wenyue Hua, Mingyu Jin, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. 2025. AIOS: LLM Agent Operating System. arXiv:2403.16971 [cs.OS] https://arxiv.org/abs/2403.16971

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

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

[44] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560 [cs.AI] https://arxiv.org/abs/2310.08560

[45] Tian Pan, Enge Song, Yueshang Zuo, Shaokai Zhang, Yang Song, Jiangu Zhao, Wengang Hou, Jianyuan Lu, Xiaoqing Sun, Shize Zhang, Ye Yang, Jiao Zhang, Tao Huang, Biao Lyu, Xing Li, Rong Wen, Zhigang Zong, and Shunmin Zhu. 2025. Hermes: Enhancing Layer-7 Cloud Load Balancers with Userspace-Directed I/O Event Notification. In *Proceedings of the ACM SIGCOMM 2025 Conference* (Coimbra, Portugal) *(SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 363–380. doi:10.1145/3718958.3750469

[46] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: Large Language Model Connected with Massive APIs. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 126544–126565. doi:10.52202/079017-4020

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

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

[49] Zhangcheng Qiang, Weiqing Wang, and Kerry Taylor. 2024. Agent-OM: Leveraging LLM Agents for Ontology Matching. *Proc. VLDB Endow.* 18, 3 (Nov. 2024), 516–529. doi:10.14778/3712221.3712222

[50] 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)*. USENIX Association, Santa Clara, CA, 155–170. https://www.usenix.org/conference/fast25/presentation/qin

[51] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *International Conference on Learning Representations*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (Eds.), Vol. 2024. 9695–9717. https://proceedings.iclr.cc/paper_files/paper/2024/file/28e50ee5b72e90b50e7196fde8ea260e-Paper-Conference.pdf

[52] 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)*. USENIX Association, Renton, WA, 2407–2421. https://www.usenix.org/conference/nsdi26/presentation/ruan-cortex

[53] 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)*. USENIX Association, Boston, MA, 969–985. https://www.usenix.org/conference/osdi23/presentation/saokar

[54] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 68539–68551. doi:10.52202/075280-2997

[55] Souhaila Serbout and Cesare Pautasso. 2024. APIstic: A Large Collection of OpenAPI Metrics. In *Proceedings of the 21st International Conference on Mining Software Repositories* (Lisbon, Portugal) *(MSR '24)*. Association for Computing Machinery, New York, NY, USA, 265–277. doi:10.1145/3643991.3644932

[56] Swagger. 2014. *Swagger RESTful API Documentation Specification.* https://docs.swagger.io/spec.html Accessed: 2026-01-06.

[57] 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* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1302–1316. doi:10.1145/3676641.3716278

[58] 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.

[59] 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.

[60] Chenxu Wang, Xumiao Zhang, Runwei Lu, Xianshang Lin, Xuan Zeng, Xinlei Zhang, Zhe An, Gongwei Wu, Jiaqi Gao, Chen Tian, Guihai Chen, Guyue Liu, Yuhong Liao, Tao Lin, Dennis Cai, and Ennan Zhai. 2025. Towards LLM-Based Failure Localization in Production-Scale Networks. In *Proceedings of the ACM SIGCOMM 2025 Conference* (Coimbra, Portugal) *(SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 496–511. doi:10.1145/3718958.3750505

[61] Haopei Wang, Anubhavnidhi Abhashkumar, Changyu Lin, Tianrong Zhang, Xiaoming Gu, Ning Ma, Chang Wu, Songlin Liu, Wei Zhou, Yongbin Dong, Weirong Jiang, and Yi Wang. 2024. NetAssistant: Dialogue Based Network Diagnosis in Data Center Networks. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 2011–2024. https://www.usenix.org/conference/nsdi24/presentation/wang-haopei

[62] Shujun Wang, Yongqiang Tian, and Dengcheng He. 2023. gDoc: Automatic Generation of Structured API Documentation. In *Companion Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23 Companion)*. Association for Computing Machinery, New York, NY, USA, 53–56. doi:10.1145/3543873.3587310

[63] Shujun Wang, Yongqiang Tian, and Dengcheng He. 2023. Improving API Documentation Comprehensibility via Continuous Optimization and Multilingual SDK. arXiv:2303.13828 [cs.SE] https://arxiv.org/abs/2303.13828

[64] Zhaodong Wang, Samuel Lin, Guanqing Yan, Soudeh Ghorbani, Minlan Yu, Jiawei Zhou, Nathan Hu, Lopa Baruah, Sam Peters, Srikanth Kamath, Jerry Yang, and Ying Zhang. 2025. Intent-Driven Network Management with Multi-Agent LLMs: The Confucius Framework. In *Proceedings of the ACM SIGCOMM 2025 Conference* (Coimbra, Portugal) *(SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 347–362. doi:10.1145/3718958.3750537

[65] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 24824–24837. doi:10.52202/068431-1800

[66] 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)*. USENIX Association, Santa Clara, CA, 715–730. https://www.usenix.org/conference/atc24/presentation/wei

[67] 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)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 26092–26130. doi:10.18653/v1/2025.acl-long.1266

[68] 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* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 661–678. doi:10.1145/3651890.3672268

[69] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. A-MEM: Agentic Memory for LLM Agents. arXiv:2502.12110 [cs.CL] https://arxiv.org/abs/2502.12110

[70] Yingxuan Yang, Huacan Chai, Yuanyi Song, Siyuan Qi, Muning Wen, Ning Li, Junwei Liao, Haoyi Hu, Jianghao Lin, Gaowei Chang, Weiwen Liu, Ying Wen, Yong Yu, and Weinan Zhang. 2025. A Survey of AI Agent Protocols. arXiv:2504.16736 [cs.AI] https://arxiv.org/abs/2504.16736

[71] 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. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 11809–11822. doi:10.52202/075280-0517

[72] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629

[73] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 62557–62583. doi:10.52202/079017-2000

[74] 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 Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'24/IAAI'24/EAAI'24)*. AAAI Press, Article 2198, 8 pages. doi:10.1609/aaai.v38i17.29946