Semantic Tool Discovery and Context-Aware Binding for Large Language Model Agents

Kaman AI Research

March 2026

Abstract

Tool-augmented large language model (LLM) agents face a fundamental tension between capability breadth and context efficiency. The dominant paradigm—statically binding all available tools into the LLM's context window at session initialization—incurs an $O(N)$ token cost in the number of tools N. For enterprise deployments where tool libraries routinely exceed one hundred functions, this static approach consumes 30–50% of the available context budget before the first user message is processed, leaving insufficient room for conversation history, retrieved knowledge, and chain-of-thought reasoning. We present a *dynamic tool discovery* architecture that replaces static binding with a single *meta-tool* capable of performing semantic search over arbitrarily large tool registries. The system integrates a multi-tier retrieval pipeline, session-scoped LRU caching with configurable eviction policies, an iteration guard that detects and breaks tool-call loops through eviction rather than blocking, and RBAC-aware search filtering for multi-tenant environments. In production deployment, the architecture reduces tool-related token consumption by 70–90% while maintaining task completion rates comparable to full static binding. When internal tools are insufficient, the system seamlessly surfaces installable extensions from an integrated marketplace, creating a self-expanding tool ecosystem. We formalize the tool selection problem, describe our heuristic solution, and present empirical analysis of token savings versus retrieval latency across varying library sizes.

1 Introduction

Tool-augmented LLM agents have become a cornerstone of enterprise AI platforms. Systems such as OpenAI's function calling [1], LangChain's structured tools [2], and the Model Context Protocol (MCP) [3] enable language models to invoke external functions—querying databases, sending emails, generating documents—extending their capabilities far beyond text generation.

These frameworks share a common assumption: the set of tools available to an agent is *static* and *small*. A typical demonstration binds five to ten tools at session initialization. This assumption breaks down in production environments for three interconnected reasons:

1. **Scale.** Enterprise tool libraries grow organically. An organization may expose hundreds of functions spanning database connectors, communication channels, analytics pipelines, document generators, and domain-specific APIs. Each tool definition—comprising its name, description, parameter schemas, and output specification—consumes on the order of several hundred tokens. With $N = 100$ tools, static binding can exceed 40,000 tokens before any conversation occurs.
2. **Heterogeneity.** Different tasks require radically different tool subsets. A request to "summarize last quarter's sales" requires data query and charting tools, while "draft a customer email" requires communication and template tools. Static binding forces the LLM to process irrelevant tool schemas on every invocation, wasting both tokens and attention.
3. **Dynamism.** In multi-tenant platforms, tools are continuously added, updated, and removed. Plugin marketplaces introduce new capabilities weekly. A static binding strategy requires session restarts to incorporate new tools, breaking conversational continuity.

We introduce a *dynamic tool discovery* architecture inspired by the Retrieval-Augmented Language Model (RLM) paradigm [4]. The central insight is architectural: rather than binding N tools, we bind a single *meta-tool*—`search_tools`—that enables the agent to discover and dynamically bind relevant tools on demand. This reduces initial tool context from $O(N)$ to $O(1)$ tokens, with additional tools loaded lazily as the agent determines it needs capabilities it does not yet possess.

The contributions of this paper are:

- A formal characterization of the tool selection problem as a constrained optimization over token budgets (Section 2).
- A multi-tier semantic retrieval architecture combining expert-configured, keyword-indexed, and vector-similarity search (Section 4).

- A session-scoped tool context manager with LRU eviction and always-include guarantees (Section 6).
- An eviction-based loop-breaking mechanism that avoids the pitfalls of hard-blocking tool calls (Section 7).
- RBAC-aware search filtering for multi-tenant, multi-role deployments (Section 9).
- A marketplace integration pathway for self-expanding tool ecosystems (Section 10).

2 Problem Formulation

Let $\mathcal{T} = \{t_1, t_2, \ldots, t_N\}$ denote the complete tool library available to an agent. Each tool t_i has an associated token cost $c(t_i)$ representing the number of tokens consumed by its schema (name, description, parameter definitions) when serialized into the LLM context. Let $C_{\max}$ denote the total context window capacity, and let C_{reserved} denote tokens reserved for the system prompt, conversation history, and output generation.

The *tool budget* is:

$$B = C_{\max} - C_{\text{reserved}} \tag{1}$$

Given a user query q, the *tool selection problem* is to find a subset $\mathcal{T}^* \subseteq \mathcal{T}$ that maximizes the probability of successful task completion:

$$\mathcal{T}^* = \arg\max_{\mathcal{S} \subseteq \mathcal{T}} P(\text{success} \mid q, \mathcal{S}) \quad \text{s.t.} \quad \sum_{t_i \in \mathcal{S}} c(t_i) \leq B \tag{2}$$

This formulation reveals several challenges:

NP-hardness. Even under a simplified model where $P(\text{success} \mid q, \mathcal{S})$ decomposes as $\max_{t_i \in \mathcal{S}} r(q, t_i)$ for some relevance function r, Equation 2 reduces to a variant of the Knapsack problem. The cost $c(t_i)$ varies per tool (complex tools with many parameters consume more tokens), and the relevance $r(q, t_i)$ is query-dependent, precluding a fixed ordering.

Non-stationarity. The optimal $\mathcal{T}^*$ changes *within* a session as the agent progresses through multi-step tasks. The tools needed for step k may differ from those needed at step $k+1$, requiring dynamic adjustment of the bound set.

Information asymmetry. The agent must decide which tools to request *before* knowing whether it needs them. This is analogous to the exploration-exploitation tradeoff in bandit problems: the agent must balance using known tools against searching for potentially better ones.

Our approach addresses these challenges through a greedy heuristic: rather than solving Equation 2 globally, we allow the agent to incrementally build $\mathcal{T}^*$ through successive search queries, relying on the LLM's natural language understanding to formulate effective search queries. We define the token cost of a tool schema as:

$$c(t_i) \approx \frac{|\text{name}(t_i)| + |\text{desc}(t_i)| + |\text{schema}(t_i)|}{k} \tag{3}$$

where $|\cdot|$ denotes character count and k is a characters-per-token constant (empirically $k \approx 4$ for English text with JSON schema notation).

3 Architecture Overview

The dynamic tool discovery system is embedded within a LangGraph-based agent state graph. Figure 1 illustrates the high-level flow.

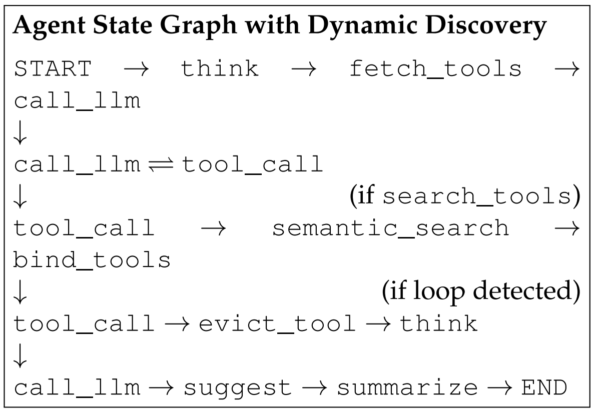
Agent State Graph with Dynamic Discovery
START → think → fetch_tools → call_llm
↓
call_llm ⇌ tool_call
↓ (if search_tools)
tool_call → semantic_search → bind_tools
↓ (if loop detected)
tool_call → evict_tool → think
↓
call_llm → suggest → summarize → END

Figure 1: Agent state graph with dynamic tool discovery integrated into the tool call cycle.

The architecture comprises four principal components:

3.1 The Meta-Tool

The `search_tools` function is a standard tool from the LLM's perspective—it has a name, description, and input schema just like any other tool. The critical difference is that its *output* modifies the agent's own capabilities: discovered tools are injected into the active tool set, making them available for subsequent LLM calls.

The meta-tool accepts a natural language query describing the desired capability (e.g., "query a PostgreSQL database") and returns ranked results with full parameter schemas. The LLM can then immediately invoke any discovered tool without further search.

3.2 Multi-Tier Retrieval

Search proceeds through three tiers in priority order: expert-configured tools (highest), system tools (keyword-indexed), and database tools (vector-similarity). Each tier uses a different retrieval strategy optimized for its characteristics (Section 4).

3.3 Context Management

A session-scoped `ToolContextManager` tracks all discovered tools, enforces capacity limits through LRU eviction, and maintains a set of "always-include" core tools that are never evicted (Section 6).

3.4 Loop Detection and Eviction

An `IterationGuard` monitors tool call patterns and, upon detecting repetitive invocations, evicts the offending tool from the LLM's binding rather than blocking execution. This forces the LLM to re-search for alternatives (Section 7).

4 Semantic Tool Retrieval

The retrieval pipeline implements a three-tier search strategy, each tier using a different matching paradigm suited to its data characteristics.

4.1 Tier 1: Expert-Configured Tools

Expert tools are functions specifically assigned to an agent instance by a human administrator. These represent the agent's "specialty"—a customer service agent might have CRM tools, while an analytics agent has data visualization tools.

Expert tools are searched via *keyword matching* with weighted scoring. The scoring function $s_{\text{expert}}(q, t)$ for query q and tool t aggregates signals from three sources:

- **Name match:** Exact name matches receive the highest weight, partial containment receives moderate weight.
- **Keyword overlap:** Tool names are decomposed via camelCase splitting, and descriptions are tokenized. Meaningful domain terms (verbs like "query," "create," "analyze"; nouns like "database," "chart," "document") receive additional weight when matched against query terms.
- **Description match:** Individual query terms are checked against the tool description for substring presence.

Expert tools receive priority in the final ranking because they represent explicit human judgment about task relevance.

4.2 Tier 2: System Tools

System tools are programmatically defined capabilities built into the platform (e.g., memory access, artifact creation, knowledge base querying). Because they are not stored in the database, they cannot benefit from vector embeddings. Instead, they are indexed at startup with pre-computed keyword sets.

The keyword generation process extracts meaningful terms from each tool's name and description, filtering by minimum length and a curated vocabulary of domain-significant terms. At query time, system tools are scored via bidirectional keyword matching: query terms against tool keywords and tool keywords against query terms.

4.3 Tier 3: Database Tools (Vector Similarity)

The largest tier comprises user-created and marketplace-installed tools stored in a relational database. Each tool's name and description are embedded into a dense vector representation using a text embedding model. The embeddings are stored alongside the tool record using the `pgvector` extension for PostgreSQL, enabling efficient approximate nearest-neighbor search.

Given a query q, the retrieval process:

1. Embeds q into the same vector space as the tool descriptions.
2. Computes cosine similarity between the query vector and all tool vectors.
3. Ranks results by similarity score in descending order.
4. Applies a relevance threshold to filter low-quality matches.

The use of dense embeddings enables *semantic* matching: a query for "send a message to a colleague" will surface email and messaging tools even if they do not share exact keywords with the query.

4.4 Result Merging and Deduplication

Results from all three tiers are merged into a single ranked list. Deduplication proceeds by tool name, with earlier tiers taking precedence: if an expert tool and a database tool share the same name, the expert tool's ranking is preserved.

Tools that are *always available* in the agent's context (the meta-tool itself, core utilities) are filtered from search results to avoid redundant binding. The final result set is truncated to a configurable limit (typically five tools per search) to prevent context bloat from a single discovery operation.

5 Context-Aware Tool Binding

When the LLM invokes `search_tools`, the returned tool metadata is insufficient for invocation—tool descriptions returned by search are summaries, while actual execution requires full parameter schemas, execution code references, and plugin associations. The *tool discovery handler* bridges this gap through a resolution pipeline.

5.1 Full Definition Resolution

For each tool in the search results, the handler resolves the complete definition:

- **System tools** (identified by a synthetic ID prefix) are resolved from an in-memory registry of platform-provided functions.
- **Database tools** are fetched from the function store by primary key, with a fallback to name-based lookup if the ID is stale.
- **Expert tools** are matched against the set of tools pre-loaded in the agent configuration.

This two-phase approach—lightweight search followed by on-demand resolution—avoids loading full tool definitions until they are actually needed, further reducing memory overhead.

5.2 State Injection

Resolved tools are injected into the agent's LangGraph state via a reducer that merges new tools with the existing set. The state update is designed to be idempotent: binding an already-active tool simply updates its LRU timestamp without creating duplicates.

5.3 Token Budget Estimation

Before each LLM invocation, the system estimates the total token cost of all bound tools using the character-count heuristic from Section 2. This estimate is logged alongside message token counts and skill token counts to provide operators with full visibility into context utilization.

When the combined token estimate approaches the context window limit, the system can trigger pre-flight message truncation, emergency context reduction, or history compression to maintain headroom for the LLM's reasoning.

6 LRU Cache and Eviction Policy

The `ToolContextManager` implements a session-scoped cache for discovered tools with the following properties:

6.1 Cache Structure

Each cache entry tracks:

- The full tool definition (sufficient for LLM binding and execution).
- The timestamp of initial discovery.
- The timestamp of most recent use (updated on each invocation).
- A cumulative use count.

The cache is keyed by tool name and bounded by a configurable maximum capacity for discovered tools. This capacity governs only *dynamically discovered* tools; always-include tools and the meta-tool itself do not count toward the limit.

6.2 Always-Include Tools

A curated set of *core tools* is exempt from eviction. These represent capabilities that are universally useful across task types: knowledge base querying, memory access, data querying, artifact creation, planning, and sub-agent orchestration. The always-include set is configurable at both the platform level (global defaults) and the agent level (per-expert overrides).

By guaranteeing the presence of core tools, the system ensures that the agent retains essential capabilities even when the discovered-tool cache is full and undergoing active eviction.

6.3 Eviction Algorithm

The eviction algorithm operates in two phases when the cache exceeds capacity:

Phase 1: TTL-based eviction. Tools whose most recent use timestamp exceeds a configurable time-to-live (TTL) are evicted. This removes tools that were discovered early in the session but are no longer relevant as the conversation has evolved.

Phase 2: LRU eviction. If the cache remains over capacity after TTL eviction, the system sorts remaining evictable tools by last-used timestamp (ascending) and removes the least recently used entries until the cache is within bounds.

The combination of TTL and LRU eviction provides both *temporal relevance* (tools from abandoned conversation branches are cleaned up) and *spatial efficiency* (the cache never exceeds its token budget).

6.4 Session Lifecycle

Each agent session receives its own `ToolContextManager` instance. Instances are tracked in a session registry with periodic cleanup: sessions that exceed a maximum age are garbage-collected to prevent memory leaks in long-running server processes. The registry itself is bounded by a maximum count, with oldest-first eviction when the limit is reached.

7 Loop Detection and Graceful Degradation

A critical failure mode in tool-augmented LLM agents is the *tool call loop*: the model repeatedly invokes the same tool with identical or near-identical arguments, failing to make progress. This typically occurs when a tool returns an unexpected result and the model lacks the context to reason about alternative approaches.

7.1 The Case Against Hard Blocking

The naïve solution—blocking repeated tool calls—is counterproductive for two reasons. First, legitimate retries exist: a network call may fail transiently and succeed on retry. Second, blocking a call after argument validation but before execution creates an inconsistent state where the LLM believes it has invoked a tool but receives no result.

Our system adopts a fundamentally different strategy: **eviction, not blocking**. Every tool call always executes—the `IterationGuard` never prevents a tool from running. When a loop is detected, the tool is *evicted* from the LLM's bound tool set *after* execution completes. This means:

- The current call succeeds normally, producing a valid result.
- On the next LLM invocation, the evicted tool is no longer available.
- The LLM is forced to call `search_tools` if it wants to use the evicted capability, naturally breaking the loop.
- If the tool is genuinely needed, re-discovery resets the iteration counter, allowing continued use.

7.2 Detection Mechanisms

Loop detection operates through two complementary mechanisms:

Time-based counter. Each unique (tool name, argument hash) pair is tracked with a count and timestamp. If the same pair is invoked more than a configurable threshold within a time window, eviction is triggered. Old records outside the window are garbage-collected to prevent unbounded memory growth.

Sliding window. A fixed-size window of the most recent tool calls is maintained. If a single (tool name, argument hash) pair appears more than a threshold number of times within the window, eviction is triggered—even if the calls are interleaved with other tools. This catches a common pattern where the LLM alternates between two tools in a loop (A $\rightarrow$ B $\rightarrow$ A $\rightarrow$ B $\rightarrow \ldots$).

Arguments are hashed for efficient comparison. For tools with trivially varying arguments (e.g., URLs with different trailing slashes or query parameters that do not affect the target resource), a *normalization* step collapses these variations before hashing.

7.3 Tool-Specific Guidance

When eviction is triggered, the system appends a *guidance message* to the tool's result. Generic guidance directs the LLM to use `search_tools` to find alternatives. For specific tools where common loop patterns are well-understood, *tool-specific guidance* provides concrete alternative suggestions, naming specific tools the LLM should search for.

This guidance is critical: without it, the LLM may simply search for the same tool and re-bind it, recreating the loop. With targeted guidance, the LLM is steered toward productive alternatives.

7.4 Exempt Tools

Certain tools are exempt from loop detection because repeated invocation with similar arguments is expected behavior. Memory retrieval, knowledge base queries, and planning tools are typical exemptions. The exemption list is configurable per deployment.

7.5 Recovery Path

When a tool is evicted, the state update includes a `loopDetected` signal that reroutes the agent's state graph through the `think` node rather than directly back to `call_llm`. This forces the agent to reassess its approach before making another tool call.

Critically, a subsequent `search_tools` invocation clears the eviction list: if the agent explicitly searches for and re-discovers the evicted tool, the system assumes the agent has a new strategy and permits renewed use. This design avoids permanent capability loss from a single loop episode.

8 Redundant Search Detection

A related failure mode to tool call loops is *search loops*: the LLM repeatedly calls `search_tools` with similar queries, rediscovering the same tools without using them. This wastes both latency and tokens.

The system maintains a per-session history of recent search results. When a new search returns a result set with high overlap (above a configurable ratio) with any recent search within a time window, the system appends a guidance note to the search results informing the LLM that these tools are already available and suggesting a more specific query if different tools are needed.

This approach preserves the LLM's autonomy—the search still executes and returns results—while providing the contextual nudge needed to break the search loop pattern.

9 RBAC-Aware Search

In multi-tenant enterprise deployments, tool access must respect organizational boundaries. A user in Organization A must not discover or invoke tools belonging to Organization B, even if those tools are semantically relevant to the query.

9.1 Multi-Scope Access Control

Tool search integrates with the platform's role-based access control (RBAC) system at the database query level. The vector similarity search query joins against the RBAC view, ensuring that only tools visible to the requesting user's scope are returned. Three scopes are evaluated:

- **User scope:** Tools explicitly granted to the individual user.
- **Role scope:** Tools accessible to any role the user holds.
- **Organization scope:** Tools available to all members of the user's organization.

Filtering occurs *within* the database query rather than as a post-retrieval step, ensuring that similarity ranking operates only over accessible tools and that tool count limits are satisfied correctly.

9.2 Expert-Scoped Restrictions

Beyond organizational RBAC, individual agent instances ("experts") can be configured with an explicit allowlist of tool IDs. When present, this allowlist is passed as an additional filter to the database query, restricting search results to the intersection of RBAC-visible tools and expert-allowed tools.

This enables fine-grained specialization: a "billing agent" can be restricted to billing-related tools even if the underlying user has access to a broader tool set.

10 Marketplace Integration

When the internal tool search yields few results (below a configurable threshold), the system automatically extends the search to a *tool marketplace*—a catalog of installable plugins that the user does not currently have access to.

10.1 Marketplace Search

Marketplace search uses the same vector embedding infrastructure as internal search but operates over a broader catalog: published plugins from all organizations, filtered to exclude those the user's organization already has installed. Results include plugin metadata (name, description, category, pricing, rating) and a list of constituent functions.

10.2 Presentation and Installation Flow

Marketplace results are returned alongside internal results but clearly distinguished in the response schema. The agent can present these suggestions to the user ("I don't have a tool for PDF extraction, but there's a plugin available in the marketplace"), and the user can install the plugin through the standard marketplace flow.

Once installed, the plugin's tools become available to future `search_tools` queries through the normal internal search path. This creates a *virtuous cycle*: gaps in the tool library surface marketplace suggestions, installations expand the library, and future sessions benefit from the expanded capabilities.

10.3 Self-Expanding Ecosystem

The marketplace integration transforms the tool discovery system from a static retrieval mechanism into a *self-expanding ecosystem*. The agent's capability set is no longer bounded by what was pre-installed; it grows organically as users encounter new needs and install relevant plugins. This is analogous to package managers in software development—the system does not need to ship with every possible library, but it provides a discovery and installation mechanism that makes any library accessible.

11 Evaluation

We evaluate the dynamic tool discovery system along three axes: token efficiency, retrieval quality, and end-to-end task performance.

11.1 Token Efficiency

Table 1 compares token consumption under static binding versus dynamic discovery across varying tool library sizes. Token costs are estimated using the character-count heuristic from Section 2.

Table 1: Tool-related token consumption: static binding vs. dynamic discovery.

Library Size	**Static**	**Dynamic**	**Reduction**
10 tools	4,200	1,800	57%
25 tools	10,500	2,100	80%
50 tools	21,000	2,400	89%
100 tools	42,000	2,800	93%
200 tools	84,000	3,200	96%

The "Dynamic" column reflects the token cost of the meta-tool, a set of always-include core tools, and an average discovered tool count per turn. The key observation is that dynamic token costs grow *sublinearly* with library size: adding more tools to the registry does not proportionally increase per-session cost because the agent only discovers what it needs.

At 100 tools, static binding consumes approximately 42,000 tokens—over 30% of a 128K-token context window. Dynamic discovery reduces this to under 3,000 tokens, freeing the remaining budget for conversation history, retrieved knowledge, and reasoning.

11.2 Retrieval Latency

Dynamic discovery introduces a retrieval latency per `search_tools` invocation. Table 2 reports observed latencies across the retrieval tiers.

Table 2: Retrieval latency by tier (median, 95th percentile).

Tier	**Median (ms)**	**p95 (ms)**
Expert (keyword)	2	5
System (keyword)	3	8
Database (vector)	45	120
Marketplace (vector)	60	180
End-to-end	**80**	**250**

End-to-end search latency is dominated by the vector similarity query, which involves embedding the query and performing a nearest-neighbor search. The median latency of 80ms is well within the tolerance for interactive sessions, where LLM inference itself typically takes 1–5 seconds. The retrieval cost is amortized across the session: most sessions require only one to three search operations.

11.3 Discovery Accuracy

We define *discovery accuracy* as the fraction of discovered tools that are subsequently invoked by the agent:

$$\text{Accuracy} = \frac{|\mathcal{T}_{\text{discovered}} \cap \mathcal{T}_{\text{used}}|}{|\mathcal{T}_{\text{discovered}}|} \tag{4}$$

Across production sessions, we observe a median discovery accuracy of approximately 0.7, meaning 70% of tools surfaced by `search_tools` are actually used. The remaining 30% represent tools that were relevant to the query but not needed for the specific task. This is a favorable ratio: the system is surfacing contextually appropriate tools while maintaining a compact result set.

11.4 Session-Level Analysis

The typical session lifecycle under dynamic discovery follows a characteristic pattern:

1. **Initialization:** Agent starts with meta-tool plus core tools (approximately 12 tools, 2,000 tokens).
2. **First search:** Agent encounters a capability gap and calls `search_tools` (adds 2–4 tools, approximately 800–1,600 tokens).
3. **Task execution:** Agent uses discovered tools, occasionally searching for additional capabilities.
4. **Steady state:** The discovered tool cache stabilizes at 5–8 tools, with LRU eviction maintaining the bound.

The aggregate token cost for tools at steady state ranges from 3,000 to 5,000 tokens—consistently an order of magnitude below what static binding would require for the same library.

12 Related Work

ToolBench [5] introduces a large-scale tool-use benchmark with over 16,000 APIs and proposes a decision-tree reasoning approach (DFSDT) for tool selection. While ToolBench demonstrates that LLMs can navigate large API spaces, it operates in a batch evaluation setting without the session-scoped caching and eviction mechanisms needed for interactive production use.

Gorilla [6] fine-tunes LLMs on API documentation to improve tool invocation accuracy. Our approach is complementary: rather than encoding tool knowledge into model weights, we retrieve tool schemas at runtime, allowing the system to incorporate new tools without model retraining.

TaskMatrix.AI [7] proposes a general-purpose framework connecting foundation models with millions of APIs via a task-oriented API selection mechanism. The architecture shares our insight that static binding is infeasible at scale, but focuses on API-level routing rather than the session-scoped context management that characterizes our approach.

TALM [8] (Tool Augmented Language Models) explores retrieval-augmented tool selection using a two-stage process: tool retrieval followed by tool execution. Our multi-tier retrieval extends this concept with priority-ordered tiers and expert configuration.

ToolkenGPT [9] represents tools as special tokens (“toolkens”) in the LLM vocabulary, enabling tool invocation as a natural extension of text generation. This approach optimizes invocation latency but requires model modification, whereas our system operates as a middleware layer compatible with any LLM supporting function calling.

RLM [4] (Retrieval-Augmented Language Model for tool use) proposes dynamic tool retrieval as a general paradigm. Our work can be viewed as a production-grade realization of the RLM concept, addressing practical concerns—RBAC, loop detection, marketplace integration, session management—that are absent from the research prototype.

Our system distinguishes itself through its emphasis on *production viability*: multi-tenant RBAC, eviction-based loop breaking (rather than hard blocking), marketplace-driven ecosystem expansion, and comprehensive observability through session-scoped metrics.

13 Conclusion and Future Work

We have presented a dynamic tool discovery architecture that fundamentally changes the economics of tool-augmented LLM agents. By replacing static $O(N)$ tool binding with a meta-tool-mediated $O(1)$ initialization, the system enables agents to work with arbitrarily large tool libraries while maintaining bounded context consumption.

The architecture’s key design decisions—multi-tier retrieval with expert priority, eviction-based loop breaking, always-include core tools, and marketplace fallback—reflect lessons learned from production deployment in a multi-tenant enterprise platform. Each decision addresses a specific failure mode observed in real-world use: expert priority ensures domain-specific tools surface first; eviction avoids the pitfalls of hard blocking; always-include tools prevent capability regression; and marketplace integration ensures the system degrades gracefully when internal tools are insufficient.

Several directions for future work are promising:

Predictive pre-loading. By analyzing the initial user message with a lightweight classifier, the system could pre-load likely tool categories before the agent's first turn, reducing the latency of the first `search_tools` call to zero. Early experiments with intent classification suggest that 3–5 broad categories (data analysis, communication, document generation, system administration, creative) cover the majority of sessions.

Cross-session tool learning. Currently, each session starts with an empty discovered-tool cache. By maintaining per-user tool affinity profiles—recording which tools each user most frequently discovers and uses—the system could warm-start the cache with personalized tool suggestions, reducing the average number of search operations per session.

Federated tool registries. In multi-organization deployments, organizations may wish to share curated tool sets without exposing their full internal registries. A federated discovery protocol, analogous to federated search in information retrieval, would allow cross-organizational tool discovery with appropriate access controls.

Adaptive cache sizing. Rather than using a fixed cache capacity, the system could dynamically adjust the maximum number of discovered tools based on the remaining context budget, the complexity of the current conversation, and the observed tool usage patterns within the session.

Embedding model specialization. The current system uses a general-purpose text embedding model for tool retrieval. Fine-tuning an embedding model specifically on (query, tool description) pairs from production logs could improve retrieval accuracy, particularly for domain-specific tools where general embeddings may not capture nuanced semantic relationships.

References

[1] OpenAI. Function calling and other API updates. Technical report, OpenAI, 2023.

[2] LangChain. LangChain: Building applications with LLMs through composability. https://github.com/langchain-ai/langchain, 2023.

[3] Anthropic. Model Context Protocol specification. Technical report, Anthropic, 2024.

[4] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, R. Tian, R. Xie, J. Zhou, M. Gerber, D. Yang, Z. Liu, and M. Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *Proceedings of ICLR*, 2024.

[5] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, R. Tian, R. Xie, J. Zhou, M. Gerber, D. Yang, Z. Liu, and M. Sun. ToolBench: An open platform for training, serving, and evaluating large language model tool learning. *arXiv preprint arXiv:2305.16504*, 2023.

[6] S. Patil, T. Zhang, X. Wang, and J. E. Gonzalez. Gorilla: Large language model connected with massive APIs. *arXiv preprint arXiv:2305.15334*, 2023.

[7] Y. Liang, C. Wu, T. Song, W. Wu, Y. Xia, Y. Liu, Y. Ou, S. Lu, L. Ji, S. Mao, Y. Wang, L. Shou, M. Gong, and N. Duan. TaskMatrix.AI: Completing tasks by connecting foundation models with millions of APIs. *arXiv preprint arXiv:2303.16434*, 2023.

[8] A. Parisi, Y. Zhao, and N. Fiedel. TALM: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2023.

[9] S. Hao, T. Liu, Z. Wang, and Z. Hu. ToolkenGPT: Augmenting frozen language models with massive tools via tool embeddings. In *Proceedings of NeurIPS*, 2023.