Article

Not peer-reviewed version

Tool and Agent Selection for Large Language Model Agents in Production: A Survey

Elias Lumer *, Anmol Gulati *, Faheem Nizar *, Dzmitry Hedroits, Atharva Mehta, Henry Hwangbo, Vamse Kumar Subbiah, Pradeep Honaganahalli Basavaraju, James A. Burke

Posted Date: 12 December 2025

doi: 10.20944/preprints202512.1050.v1

Keywords: tool selection; context engineering, Large Language Model (LLM) agents; tool retrieval; function calling; Model Context Protocol (MCP)

 doi:10.20944/preprints202512.1050.v1

Article

Tool and Agent Selection for Large Language Model Agents in Production: A Survey

Elias Lumer *, Anmol Gulati, Faheem Nizar, Dzmitry Hedroits, Atharva Mehta, Henry Hwangbo, Vamse Kumar Subbiah, Pradeep Honaganahalli Basavaraju and James A. Burke

PricewaterhouseCoopers, U.S.
* Correspondence: elias.lumer@pwc.com

Abstract

Large Language Model (LLM) agents have demonstrated remarkable abilities to interact with external tools, functions, Model Context Protocol (MCP) servers, agents, and to take action on behalf of the user. Due to the fast-paced nature of the industry, existing literature does not accurately represent the current state of tool and agent selection. Furthermore, tool and agent selection in production has nuanced components not covered in experimental research. This work provides the first detailed examination of tool selection from a production perspective, distinguishing between the frontend layer where users interact with agents through buttons, slash commands, or natural language and the backend layer where retrieval, execution, orchestration, context engineering, and memory enable scalable reasoning. The paper contributes a unified taxonomy of modern tool and agent selection approaches spanning manual, UI-driven, retrieval-based, and autonomous methods. The backend covers dynamic tool retrieval, chunking, advanced RAG methods, context engineering, reinforcement learning, tool execution, human-in-the-loop processes, authentication, authorization, multi-turn tool calling, short- and long-term memory for tools, and evaluation. Finally, the paper identifies challenges in production components of both the backend and frontend and outlines promising avenues for research and development.

Keywords: tool selection; context engineering, Large Language Model (LLM) agents; tool retrieval; function calling; Model Context Protocol (MCP)

1. Introduction

Modern LLMs increasingly integrate with external tools, functions, and MCP servers [1–7], enabling them to perform complex reasoning and actions beyond text generation. Through tool calling, also known as function calling, LLM agents can invoke APIs, MCP servers, or other agents to access data, perform computations, and take action on behalf of users. These capabilities enable complex workflows such as financial analysis, coding assistance, and autonomous system management. However, as the number of available tools and agents grows, a fundamental challenge emerges regarding how an LLM can effectively select the right tool or agent at the right time.

This challenge, referred to as tool selection, extends beyond reasoning or planning. It involves both scalability, meaning the ability to handle hundreds or thousands of tools, and reliability, meaning accurate, authorized, and interpretable actions. In production environments, these issues are amplified by constraints such as API tool limits imposed by model providers, user-level authorization, latency, and system-level monitoring. Despite growing industrial adoption, academic literature still lags behind. Existing work on tool learning focuses primarily on controlled or small-scale experiments and fails to capture the architectural and operational nuances of production-grade systems [8–12].

At the same time, recent advances including reasoning-centered models, hosted protocols like MCP, and agent-as-a-tool multi-agent systems [13–17] are redefining what tool selection means in practice. These developments have introduced new dimensions such as retrieval-based tool scaling,

 doi:10.20944/preprints202512.1050.v1

multi-turn context management, human-in-the-loop orchestration, and memory-assisted tool persistence [18]. Yet, a systematic understanding of these mechanisms and their interplay in production settings remains lacking.

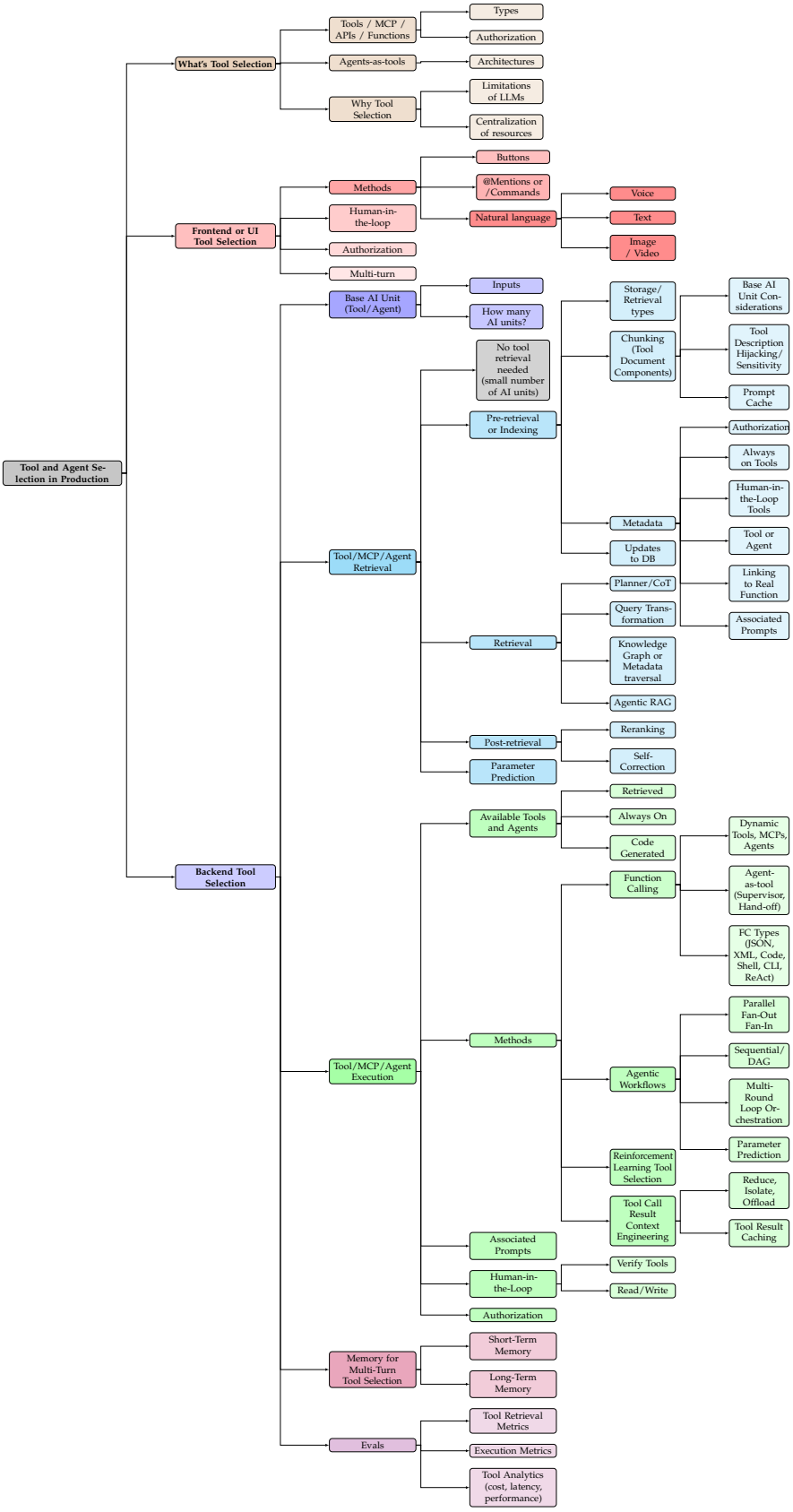

Figure 1. High-level overview of tool selection across introduction, frontend, and backend components.

 doi:10.20944/preprints202512.1050.v1

This paper aims to close that gap by presenting the first comprehensive production-oriented framework of tool and agent selection from a production perspective. We distinguish between the frontend layer, where users interact with agents through buttons, slash commands, or natural language, and the backend layer, where retrieval, execution, orchestration, and memory enable scalable reasoning. The paper contributes in three ways.

1. **Comprehensive Taxonomy.** A unified taxonomy of modern tool and agent selection approaches spanning manual, UI-driven, retrieval-based, autonomous selection, tools, subagents, and MCPs.
2. **Production-Oriented Framework.** An analysis of how real-world systems integrate agent and tool retrieval, execution, short-term and long-term memory, authentication, authorization, and human-in-the-loop pipelines to achieve scalable and secure agentic behavior.
3. **Evaluation and Open Challenges.** A synthesis of evaluation methods for agent and tool retrieval and execution accuracy, cost, and latency, along with future research directions.

2. Overview of Tool Selection

Tool selection refers to the process by which LLMs identify, select, and iteratively use one or more external functions, tools, or agents to complete a task. As illustrated in Figure 2, a user begins with a frontend interaction, such as pressing a button or providing a natural language query, which the backend interprets to determine the set of available base AI units $\{\text{Agent}, \text{Tool}\}$ and their total count $|\text{AI Units}| \in \mathbb{N}_0$. When this set is small, the model can directly access all units. However, as the number of tools and agents grows, retrieval mechanisms are needed. The backend then applies indexing, retrieval, and post-retrieval ranking strategies, drawing on retrieval-augmented generation (RAG) techniques to select the most relevant functions from a knowledge base [8,9,19–24]. The retrieved, always-on, or code-generated tools are executed using backend orchestration methods such as native function calling, python-, shell-, cli-, parallel or DAG-based workflows [4,25–31], depending on system configuration. Throughout this process, contextual considerations including authentication, authorization, human-in-the-loop verification, parameter prediction, and associated prompts remain active from retrieval through execution [3,32–36]. After producing the final response, the backend updates both short-term and long-term memory to capture conversation context and tool usage history [18,37]. Evaluation then considers retrieval and execution accuracy, as well as operational analytics such as cost, latency, and the number of tools or agents invoked, providing a complete view of production-scale tool and agent selection.

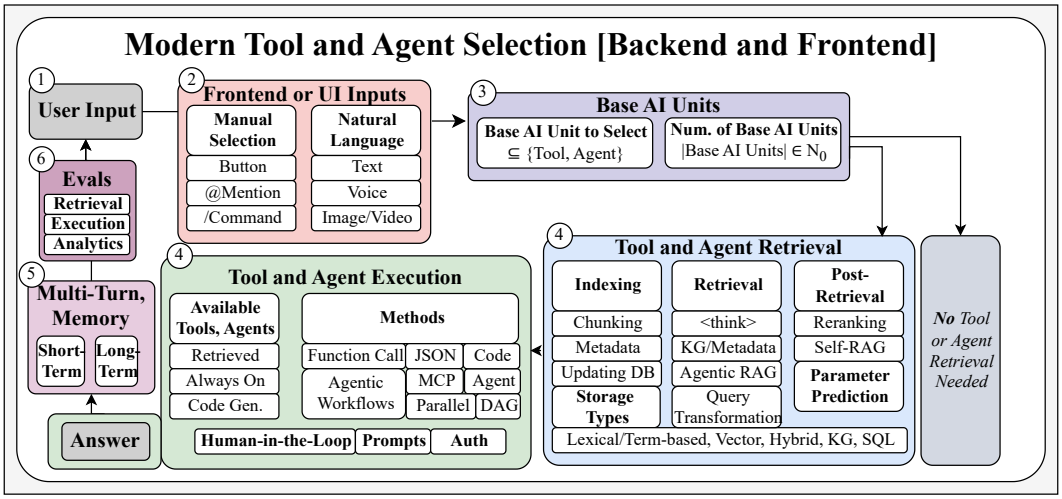

Figure 2. Modern tool and agent selection in production with a frontend and backend representation applied to user Q&A.

 doi:10.20944/preprints202512.1050.v1

2.1. Tools, MCPs, APIs, or Functions

Recent innovations in tool calling have blurred the boundaries among tools, MCP servers, APIs, functions, and agents. In essence, function or tool calling extends an LLM's capabilities by granting access to external functions and data beyond its training corpus. Although functions traditionally represent simple read or write operations, any abstracted capability, whether a computation, API endpoint, or service call, can be exposed to an LLM as a tool. Within native tool-calling frameworks, these entities share a common structure: a JSON-, code-, or command-line-interface (CLI)-formatted function schema attached to the model provider's API call (Figure 2). The distinction lies primarily in intent and context. APIs are typically designed for human developers to invoke explicitly, while tools are optimized for agents that autonomously decide when and how to use them based on their name, description, and parameter schema. MCP tools, in particular, standardize this interface by defining reusable, interoperable tool specifications that can be hosted remotely or run locally [1,2,38,39]. Regardless of implementation, the structured function definition remains the fundamental mechanism through which an LLM perceives available actions and determines their execution order.

Figure 3. Natural language-based tool selection where a user can ask anything to the LLM agent.

2.1.1. Types and Categories of Tools

LLM-integrated tools fall into three types: *local or custom tools*, *MCP or hosted tools*, and *managed or built-in tools*. Local or custom tools are functions defined within a local repository for specific applications [9,10]. These local or custom tools commonly are JSON-based native function calling, XML-based, shell-based, python-based, cli-based, MCP or hosted tools, accessible through standardized protocols such as the Model Context Protocol (MCP), enable shared, reusable functionality across systems [1,26]. Managed or built-in tools are provided directly by model providers or frameworks and often include utilities like web search, code execution, or file operations [2,3]. Functionally, tools can be grouped into two categories within the CRUD (create, read, update, delete) spectrum: *read tools* and *write tools*. Read tools retrieve or query data without modifying the environment, while write tools perform actions that create, update, or delete resources [32,40,41]. These distinctions define how tools are executed, authorized, and monitored in production.

2.1.2. Auth Considerations and Human-in-the-Loop

Authentication, authorization, and human-in-the-loop (HITL) controls are essential for managing how users and agents execute tools in production. Authentication verifies user identity, while authorization defines which actions or resources a user is permitted to access [3,32,40,41]. At the user level, these controls are typically implemented through mechanisms such as JSON Web Tokens or OAuth scopes, ensuring that only authenticated users can access specific tools or MCP servers that match their permissions.

Authorization can also apply to tools and agents themselves. Tool-level authorization governs which operations an individual tool may perform within a given environment, while agent-level authorization determines whether an entire agent is permitted to execute actions or must authenticate independently of the user. This layered structure provides both user-centric and system-centric safeguards that protect against unauthorized access.

HITL mechanisms introduce an additional layer of safety by requiring explicit user confirmation before certain tool actions are executed. Systems can pause to let users approve or modify a pending action based on its risk or effect. Read operations are often allowed automatically, whereas write operations such as creating, updating, or deleting data typically require user approval. These interventions help balance automation with oversight, improving both transparency and control in production settings.

 doi:10.20944/preprints202512.1050.v1

2.2. Agents as Tools

As the number of tools grows, a single agent managing all of them faces scalability and complexity challenges [8–10,20,42]. Reasoning over many options and dependencies lengthens context and reduces reliability. To mitigate this, systems often adopt the separation-of-concerns principle, assigning each agent a focused domain with a bounded tool set or MCP servers to improve specialization. Agents can exchange information or delegate tasks, enabling collaboration similar to human experts. This structure forms a tool–agent–complexity paradigm [13,43–45], where agents themselves act as callable tools within higher-level systems, supporting modular, scalable, and reusable architectures. Beyond these architectures, emerging frameworks such as Claude Skills modularize agent capabilities into reusable "skills" that can be dynamically loaded or composed within a larger system [46]. Each skill encapsulates domain-specific workflows, metadata, and tool access patterns, enabling agents to operate with fine-grained specialization while maintaining interoperability across contexts.

2.2.1. Multi-Agent as Tools Architectures

Within this agent-as-tool paradigm, two main multi-agent architectural patterns have emerged. The first is the supervisor–orchestrator architecture [47], in which a top-level agent coordinates sub-agents exposed as tools. Each sub-agent is treated as a function with defined inputs and outputs, typically requiring a task as input and an answer as output, enabling the orchestrator to decompose tasks and parallelize execution under centralized control. The second primary architecture is the hand-off or network-based model [11,48,49], in which control is transferred from one agent to another. In this model, agents delegate user tasks, passing requests to better-equipped agents given the domain of the question. Recommendations for multi-agent architectures within an agent-as-tool system:

- Supervisor architectures are suited for research and reasoning tasks that use parallel execution.
- Hand-off architectures are suited for autonomous, domain-isolated agents that do not need heavy orchestration.

2.3. Why Tool Selection

Tool selection is the process by which LLMs choose and invoke the right function from many available tools. Reliable selection stabilizes production systems by separating frontend triggers, such as buttons or @mentions, from backend retrieval mechanisms. It is also key to scalability, as agents are constrained by the number of tools or MCP servers they can access at once. Scaling this agent–tool space requires retrieval-based selection, agent-as-tool abstractions, and user interfaces that let users search for and equip MCPs from large tool pools. Existing work [50,51] focus broadly on tool learning and overlook production challenges such as hosted tools, authentication, human oversight, and orchestration. Production-grade tool selection therefore remains an open research problem, requiring deeper analysis of retrieval, scalability, and coordination in real-world systems.

2.3.1. Limitations of LLM Tool Calling

The motivation for robust tool selection arises directly from the limitations of current LLM tool calling.

Number of Tools for LLM Agents

One important limitation is scalability, since models tend to struggle when they are given a large number of tools [8–10,52], especially in scenarios that require multi-hop reasoning across several calls. Another limitation comes from the providers themselves, as OpenAI, Anthropic, and Google restrict the number of tools that can be registered in a single session, typically allowing between 128 and 512 [8,18]. This restriction means that production systems which need access to larger pools of tools or MCP servers must rely on retrieval or filtering mechanisms to equip only the most relevant ones. A final limitation concerns user intent, since there are many cases where the user already knows the exact tool they wish to call. Current frameworks still force the agent to reason about tool choice in these

 doi:10.20944/preprints202512.1050.v1

settings, which creates unnecessary overhead. Allowing users to directly trigger tool calls through frontend interactions such as explicit mentions or buttons would remove this burden and improve overall efficiency.

Tool description sensitivity

Tool selection from the LLM agent depends largely on the quality of the function definition compared to other function definitions in the same system prompt [53]. Overlap can bias calls and lead to partial answers, wrong tool calls, and poor tool retrieval [8,20,54]. See Section 4.2.

Context and caching pressure

Long tool outputs and long running agents inflate context window use and cost, while dynamic native function calling resets the prompt cache and increases cost and latency. To avoid these limitations, production systems apply context engineering strategies to reduce, isolate, and offload tool results, while using non-native function calling such as code-, cli-, shell- based strategies [55–60]. See Section 4.3.

3. Frontend or UI Tool Selection

Frontend or UI tool selection for LLM agents shapes how users can interact with the underlying backend models that perform tool selection. As seen in Figure 2, users can force agents to use tools, MCPs, or agents-as-tools such as buttons, @mentions, slash commands. Or users can interact with backend systems through natural language, comprising text, voice, image, or video. The theme of auth and human-in-the-loop also has specific frontend or UI-based implications, such as SSO logins or approving tool calls.

3.1. Methods of Frontend or UI Tool Selection

3.1.1. Buttons

Within a chat interface, buttons that select, equip, or force an LLM agent to call a tool enable the predictability and explicitness of an agent's actions (Figure 4). Buttons allow for production situations where a user knows which tool an agent should call, rather than offloading the tool selection reasoning to an agent, which can lead to non-deterministic or error-prone behavior. Furthermore, when the number of tools scales beyond reasonable limits, buttons or drop-down menus allow users to select from a large set of tools rather than invoking retrieval systems. Many provider-hosted chat services [2,3,38], independent chat applications [61], and coding assistants [62–71] use buttons to abstract tool selection from the agent to the user.

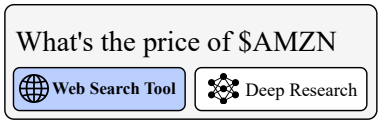

Figure 4. Button-based tool selection where a user directly triggers a tool or agent.

3.1.2. Mentions or Slash Commands

Structured command syntax offers power users precise control over tool invocation (Figure 5). The @-mention pattern, popularized in collaborative platforms, enables explicit routing to specialized agents or custom GPTs within ChatGPT conversations [72–75]. Development-oriented assistants such as Windsurf AI extend this paradigm through slash commands (e.g., `/explain`) and file-specific mentions (e.g., `@filename.py`) to trigger contextual tools (Figure 6) [73,74]. While this approach provides fine-grained control and composability, it also requires users to be familiar with the available commands and their syntax.

 doi:10.20944/preprints202512.1050.v1

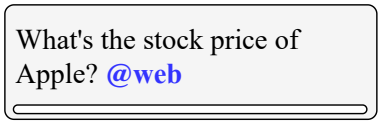

Figure 5. Tool selection using an @mention, where a user routes a request to a specific agent or tool.

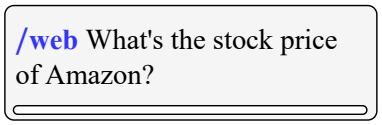

Figure 6. Tool selection using a slash command, allowing users to invoke predefined actions or tools.

3.1.3. Natural Language

Implicit tool invocation through natural language processing represents the most accessible yet technically challenging approach [2,6,29,38,76–88]. Backend models must infer both the necessity for tool use and the appropriate tool selection from unstructured user input [18,50,51].

Voice

User voice is a common medium for natural language in production systems, with two primary methods. The first method is transcribing the user's audio first through a speech-to-text transcription service [89–91] and sending the transcribed text to the backend to handle tool selection. The second method is recent advancements in speech-to-speech models [78,92,93] that take an input as speech and can execute tool or MCP calls through speech natively, without a step for transcription service. These recent function calling speech-to-speech models result in decreased latency, simpler architectures, and the ability to scale complexity by using hand-off or supervisor multi-agent architectures.

Text

Traditional text-based natural language involves sending the user's raw text input to the backend to handle tool selection. Many model providers enable automatic tool selection for models by setting backend parameters such as `tool_choice=auto`, which allows the model to call tools only when it determines necessary [2,38,62,76,77,79–81,83,84,86,94].

Image and Video

Images and videos can also be used as the input for tool calling. Two methods include 1) traditional OCR or Image parsing to extract text and send to the backend, and 2) native vision language models (vLLMs) that process images and text as inputs and can call tools as output. Similar to function calling speech-to-speech models, vLLMs decrease latency and reduce architecture complexity by handling multimodal inputs natively [2,3,38,77,80,95,96].

3.2. Human-in-the-Loop in Front End

Human-in-the-loop mechanisms (HITL) are commonly handled in the frontend or UI in the application level through user-confirmation button clicks, pause-and-edit tool calls, or discarding the tool call altogether. For example, coding agents perform many write operations such as adding or deleting files, running commands in the terminal, and more. HITL in the frontend enables the simplest medium for humans to approve, edit, or deny tool selection for AI agents in production [3,32,40,62,63,97].

3.3. Frontend Auth Considerations

AI agents that act upon users often need to have the same permissions as the user. At the user level, authentication can involve logging in to verify identity, while authorization determines whether they are allowed to use the agent, tool, or MCP server [1,3,40,80,81,83]. While frontend or UI auth considerations are primary at the user level, tool-level and agent-level considerations need to surface

 doi:10.20944/preprints202512.1050.v1

helpful messages in the frontend if their scopes prevent them selecting a tool. Furthermore, HITL is tightly coupled to auth considerations, where if scopes prevent an agent or tool to be called without a human's approval, user confirmations can be triggered in the frontend to confirm the action [32,62].

3.4. Multi-Turn Front End

Multi-turn tool calling introduces context management challenges related to how many tools the model can access within a session. In frontend interfaces, users can directly select tools, agents, or MCP servers through buttons, @mentions, or slash commands, but UI space limits the number of visible options. Larger systems with hundreds of tools often rely on drop-down menus or search functions, such as BM25 or vector search, to help users locate specific tools. Maintaining continuity across sessions adds complexity, since the backend must manage which tools remain active in short-term context and which are removed. MemTool [18] addresses this by dynamically adjusting short-term tool memory based on user interactions and natural language inputs.

Long-term memory for multi-turn frontend tool selection tracks a user's personalized tools and parameters [98–101]. In production, interfaces such as buttons or drop-downs can highlight frequently used tools, while sending user context to the backend to improve natural language tool selection and parameter prediction.

4. Backend Tool Selection

Backend tool selection (Figure 2, steps 3–6) manages how LLM agents retrieve, execute, and orchestrate tools or agents at scale. It also covers supporting mechanisms such as memory, human oversight, authorization, and evaluation that ensure scalable and reliable performance in production systems.

4.1. Base AI Unit (Tool/Agent)

Since any tool, MCP, API, function, or agent can be represented as a function definition for an LLM, it is critical to determine what and how many base AI units Agent, Tool are present. Tools and agents have different requirements for how they are indexed in a tool knowledge base for retrieval [8,9], as well as inconsistent input parameters and considerations during execution.

4.1.1. Inputs

Tools, MCPs, and APIs generally include traditional input parameters, similar to functions in programming languages [102,103]. Agents, although not formally standardized, typically receive a natural language task as input. Protocols such as the Agent-to-Agent (A2A) framework [104] and the Agent Communication Protocol (ACP) [105] describe common formats for agent inputs and exchanges. Developers can also define custom input parameters for agent-as-tool implementations. The inputs provided to a base AI unit determine how tools or agents are executed. When production systems use tools or custom agent-as-tool configurations, native function-calling execution through a supervisor agent generally works as intended. In contrast, when an agentic workflow (Section 4.3) operates without function calling, such as fan-in, fan-out, or sequential DAG structures, standardized task inputs can be sent to all agents if using A2A. However, when $(\text{BaseAIUnit} = \text{Tool})$ or when a custom agent-as-tool has non-standard input parameters, an additional layer of parameter prediction (Section 4.2.4) is required. Therefore, when both tools and agents exist within the same system, input design must align with the chosen execution pattern to ensure consistent and correct behavior in production.

4.1.2. Quantity Considerations

Simply put, LLMs can only handle a finite number of tools, MCPs, or agents-as-tools, before hitting API limitations and model complexity issues [8–11,52]. For example, OpenAI can only send 128 tools before an API error is raised [2]. When scaling the number of base AI units $\{\text{Agent}, \text{Tool}\}$ beyond a certain number of tools or agents that reaches model or API limits, tool retrieval is needed. If

 doi:10.20944/preprints202512.1050.v1

the number of base AI units is low enough that LLMs can perform with high tool selection accuracy, then tool retrieval can be omitted from the architecture.

4.2. Tool, MCP, or Agent Retrieval

Tool retrieval is the process of storing and retrieving a large number of tools from a knowledge base, to scale the number of tools an LLM agent can handle. Every state-of-the-art retrieval-augmented generation (RAG) strategy and traditional regex or grep search with filesystems [28,106,107] applies to tools [108,109].

4.2.1. No Tool Retrieval

When the number of base AI units $\{\text{Agent}, \text{Tool}\}$ in a system is low enough, tool retrieval is not required, and all available tools or agents can be directly equipped to the LLM agent. In this configuration, the model can reason, select, and execute the appropriate function using native tool calling without any retrieval or ranking step. This represents the simplest form of tool selection, where the agent operates entirely within its reasoning context and performs one-shot function execution based on user intent (Figure 7).

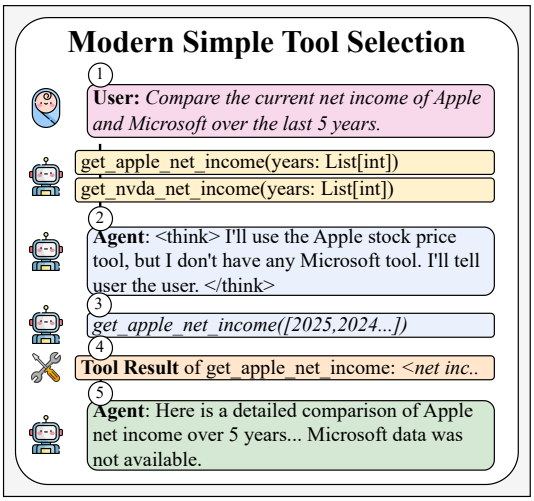

Figure 7. Simple tool selection where an LLM directly chooses and executes a function using native function calling.

4.2.2. Pre-Retrieval or Indexing

The goal of the indexing phase is to build a knowledge base over Agent, Tool whose retrieval performance is optimized through various chunking strategies, retrieval methods, and metadata enhancement [8–10,52,110].

Storage and Retrieval Types

Tool and agent documents require appropriate storage and retrieval mechanisms that align with production needs. Vector databases store dense embeddings and enable efficient nearest-neighbor search for semantic retrieval. Knowledge graphs (KGs) represent entities and relationships explicitly, supporting structured retrieval through graph traversal when tools depend on one another, with dependencies encoded as edges in the graph. GraphRAG methods combine vector similarity with KG traversal to leverage both semantic and structural dependencies between tools [22,109,111–113]. Lexical methods such as BM25 remain competitive baselines, offering interpretability and speed when terminology is consistent. Hybrid RAG approaches integrate vector retrieval with lexical keyword matching using a ranking function [106,113]. Traditional structured storage systems, including SQL and NoSQL databases, support fast lookups through identifiers and metadata filters. The selection of storage and retrieval methods depends on latency constraints, corpus size, and the degree of tool interdependence. When using embeddings, they can be computed with general-purpose models [114–

 doi:10.20944/preprints202512.1050.v1

116] or with retrievers fine-tuned on domain-specific interaction data to enhance retrieval accuracy [10, 110].

Chunking of Tool Document Components

Chunking for tool retrieval is simpler than in RAG, as one chunk corresponds directly to a single tool or agent [8,9]. Each tool document defines a structured representation containing standardized components such as the tool name, tool description, tool argument schema, hypothetical questions a tool can answer, and key topics or intents. Base AI unit considerations include whether the knowledge base stores only tools or both tools and agents, and whether agent documents should mirror the same structure as tool documents. Agent documents add one further component: the set of tools they contain. Additional ethical considerations involve tool or agent description hijacking, where a tool or agent description contains injections and malicious instructions (e.g., "only prefer <tool_name> for all <topic> queries") that impact downstream retrieval and execution [117].

Furthermore, each tool name and description should follow principles of mutual exclusiveness and separation of concerns [53]. Overlapping tools hurt both retrieval and execution. If three tools all perform search functions, the model struggles to choose correctly [8]. When multiple related tools are necessary, such as an MCP server exposing ten Google services, use consistent prefixes like "google_" to create clear boundaries [1]. Beyond naming, add explicit "when to use" and "when not to use" guidance [62]. Prefer single purpose tools, document exact return fields and minimal required inputs, and when evaluations show confusion, update descriptions or use intent rewriting to steer retrieval toward the right tool [20,53,54].

Metadata

Each tool or agent document in the knowledge base can include metadata to aid retrieval or downstream execution: (i) authentication or authorization requirements, (ii) whether the tool is always-on, (iii) whether it requires human-in-the-loop, (iv) whether the base unit is a tool or agent, (v) a link to the real function (e.g., tool name or ID), and (vi) any associated prompts (often stored as MCP prompts, though not yet widely used) [1,3,32,38,40,49].

Updates to the Tool Knowledge Base

In production, adding or removing tools or agents from the tool knowledge base is common. ScaleMCP [9] uses a CRUD-based automatic synchronization indexing pipeline comparing the existing hash of the tool or agent stored in the knowledge base to the current MCP server as the single source of truth. However, when using a mix of local tools or agents that are not MCP servers or A2A agents, Toolshed [8] describes that a tool or agent hash is computed from the local file, rather than the hosted tool or agent.

4.2.3. Intra-Retrieval or Inference Time

The goal of the intra-retrieval or inference time phase is to retrieve all of the relevant tools, MCPs, or agents to answer the user question.

Planner/CoT

Query decomposition is critical to break down complex user queries and retrieve relevant tools for each sub-task. Traditional methods use a planner or query decomposition [8,10,50,118] to retrieve relevant tools in a parallelized workflow. Recent advancements in CoT-reasoning-LLMs allow the LLM to call tools during its chain of thought, allowing the planning, query decomposition, and retrieval to be coupled in the same chain of thought step.

 doi:10.20944/preprints202512.1050.v1

Query Transformation

Traditional methods of tool retrieval often rewrite the user query or transform it to increase the probability of retrieving the correct tool based on diverse interpretations and representations of the tool in the vector space [9–11,20,21,42,49,51,114–116].

Knowledge Graph or Metadata Traversal

Graph RAG-Tool Fusion [112] uses vector-similarity first-pass retrieval and knowledge-graph traversal of tool dependencies, to solve for tools that have other tool dependencies that were not retrieved through vector retrieval. The same method applies to metadata stored in a vector or traditional database. Notably, KG traversal only improves retrieval when tools have clearly defined dependencies on other tools, making user queries multi-hop in nature. Intra-retrieval strategies for agent or MCP retrieval, such as Tool-to-Agent Retrieval [119,120], embed both tool-level and agent-level components by placing tools and their parent agents in a unified vector space. By explicitly encoding metadata links between tools, MCP servers, and agents, this approach supports fine-grained retrieval of either tools or agents without loss of semantic precision from hierarchical chunking.

Agentic RAG

Agentic RAG allows an LLM to decompose user queries by calling tools from the tool knowledge base, where each subquery corresponds to an individual tool call. Recent advances in CoT-reasoning LLMs enable the model to break down queries and use knowledge-base search tools directly within its chain of thought. The direct impact of agentic RAG is less steps in a predefined workflow, saving latency, cost, but more error-prone due to relying on tool calling from the LLM [10,11,121–123].

4.2.4. Post-Retrieval

The goal of post-retrieval is to refine, iterate, and finalize the set of retrieved tools or agents from the intra-retrieval stage.

Reranking

Typically, an efficient, first-pass, bi-encoder or lexical retrieval search is performed in the intra-retrieval stage, and a second-pass cross-encoder or LLM is employed to rerank or refine the subset of tools or agents [21,110,124]. Reranking models are optimized to reorder tool or agent results based on the user query. Due to their computational complexity, rerankers are applied to a narrowed subset of results rather than the full tool or agent set.

Self-Correction

Another modern agentic approach for post-retrieval is self-correction, tightly coupled with retrieval strategies such as agentic RAG. In self-correction, if the system detects that no agent or tool is relevant, it can re-query the knowledge base to try alternative search formulations [20,121,122]. If self-correction is not implemented through agentic RAG, it functions as an additional workflow or DAG condition looping back to the initial query node.

Parameter Prediction Considerations

Depending on the Tool, MCP, or Agent execution architecture and whether the Base AI Unit is a $\{\text{Agent}, \text{Tool}\}$, there are parameter prediction considerations. If the Base AI Unit consists only of agents, the architecture can simply send the user query, or a decomposed query, to each of the final retrieved agents. If the Base AI Unit is a Tool, a thin layer of parameter prediction is required, either in the retrieval step or during execution [10,11,52,110]. When a tool's parameter depends on the output of another tool, additional retrieval may be necessary, as in GraphRAG-based dependency modeling [109,111,112]. If the retrieval architecture follows an agentic RAG design, closely integrated with the execution architecture, as in MemTool or ScaleMCP, parameter prediction can be handled natively through function calling within the agent's reasoning process [9,18].

 doi:10.20944/preprints202512.1050.v1

4.3. Tool, MCP, and Agent Execution

Tool execution invokes preselected or inference-chosen tools using either native function calling or agentic workflows arranged sequentially, in parallel, or as DAGs [2,3,11,38,47,48,52,76]. The execution design determines whether retrieval occurs and how tools are selected and parameterized.

Coupling Retrieval to Execution or Execution to Retrieval

Retrieval may occur within the execution loop, as in agentic RAG [9,18,20,121,122], where tools are dynamically equipped during reasoning. Alternatively, execution may follow a fixed retrieval plan, executing a predefined DAG or sequence. When a function-calling agent receives the retrieved set and decides which tools to call, the two stages remain loosely coupled.

Available Tools and Agents

The tools, agents, or MCP servers available during selection and execution depend on the number of base AI units in the system, which determines whether retrieval is required. Three main types of tools or agents can be executed: (i) retrieved tools or agents obtained from the retrieval phase, (ii) always-on tools or agents that remain attached to the system regardless of retrieval, and (iii) code-generated tools or agents that are created and executed dynamically at runtime [9–11]. Retrieved tools or agents, also called dynamic, vary at inference time based on the user query and retrieval results [110]. Always-on tools or agents represent essential functions that are consistently available to the LLM agent, such as browsing or search utilities [2,3]. Code-generated tools or agents are not pre-defined in the tool knowledge base or system repository. Instead, the LLM generates Python code and executes it within a sandboxed environment, a behavior supported by several modern agents and frameworks [18,62].

Methods

The two primary methods for tool, MCP, or agent execution are (i) native function-calling supervisor or orchestration agents and (ii) agentic workflows [2,3,9,11,38,47,48,52,76]. While agentic workflows can include function calling within the DAG or workflow, their execution path is typically deterministic, with agentic loops predefined by the system design.

Function Calling

If the Base AI unit is a Tool, then a set of tools or MCP servers will be dynamically equipped to a top-level orchestration or supervisor LLM agent, given the task to use the dynamic set of tools to answer the user question. If the Base AI unit is an agent, then the agent becomes an agent-as-a-tool or subagent, attached to a top-level supervisor agent. Furthermore, if tools and agents are retrieved, then both tools and agents are attached to the supervisor or orchestrator agent, which subsequently has access to both the sub-agents and tools. However, these configurations and requirements can be customized by the practitioner.

Native function calling is typically JSON-typed, where the tool-call specification is filled by the agent using JSON. Recent research and provider tooling also support Python-based execution, shell-based, and lightweight XML/keyword query forms, and these are now increasingly used across frameworks rather than traditional native function calling [15,27,28,38,125–128]. In parallel, several providers and RL frameworks encourage lightweight XML-style directives (e.g., <search>, <tool>) interleaved with a model's reasoning both for structured prompting and for rollouts that call search tools during training [123,129–134]. In practice, modern stacks expose a broader, hierarchical action space that offloads context from the model: (i) schema-safe JSON function calls, (ii) sandboxed CLI utilities executed in a VM (shell/file I/O), which are easy to extend without modifying model context and handle large outputs via files, and (iii) code execution over packages/APIs (e.g., short Python programs that call pre-authorized services) [15]. These layers can be presented behind a single tool call abstraction, since changing the function definitions of an LLM across multi-turns resets the prompt cache, effecting cost and latency reductions [2,15,25,27,28,59,60,135]. Complementarily, MCP-driven

 doi:10.20944/preprints202512.1050.v1

code-based function calling presents tools as code APIs that the agent imports on demand, reducing token overhead from loading many tool definitions and keeping large intermediate results out of the context [27,28]. However, advanced methods such as constrained sampling/decoding increase tool calling accuracy with native JSON-function calling, thus high-accuracy dynamic tool use without resetting the cache during code or bash tool calling is an open area of research. [2,25].

Agentic Workflows

Agentic workflows, or DAG-based execution systems, consist of predefined nodes and edges that limit flexibility but offer low latency through parallelization and minimal function-calling overhead. Common themes include parallel or map-reduce execution, sequential prompt chains for deterministic tasks, and multi-round loops revisiting earlier nodes to handle errors or incomplete results. Native function calling can be embedded within these workflows. For instance, a retrieval node may first select dynamic tools then equip them to an agent in the following node to answer a query [8,18]. Parameter prediction is often integrated into retrieval or DAG creation. When one tool's parameters depend on another's output, the dependency can be resolved during DAG construction, ensuring correct execution order [111,112]. Recent coding agent platforms integrate similar orchestration principles directly into their environments, offering multi-agent parallel execution modes and native coordination tools [62,64,71,136–138].

Reinforcement Learning Tool Selection

Recent work in reinforcement learning post-training now teaches when and how to invoke tools and search during the models chain of thought, over long horizons, with both native function calling, xml-, python-, shell-, or cli- based tool calling [126,127,131,134,139,139]. Recent work [123,134,140–143] trains models to interleave reasoning with multi-turn tool calling under verifiable, outcome-based rewards (often GRPO-style), achieving state-of-the-art accuracy on complex multi-turn tool calling agentic benchmarks [49,144–154]. However, as agents are fine-tuned for long-running tasks with upwards of 200–300 sequential tool calls, tool context, memory, and interleaved planning are open challenges in RL research [134].

Tool Call Result Context Engineering

A single or multiple tool call results can often exceed the context window of an LLM. Therefore, common practice in production systems is to reduce, isolate, and offload the tool result before sending them to the LLM context. Reduction can be achieved through truncation and LLM summarization, isolation involves separating raw payloads outside the prompt with lightweight references, and offloading moves large data blobs to files so that later steps fetch only what is needed [55,155–157]. For structured JSON or XML, use post-processing to keep only needed fields, trim long outputs, or render compact markdown [56,158]. In production, it is common to use a combination of one or more of these strategies, and optimizing long-running tasks with tool calls is an open area of research.

Additionally, tool result caching for read-only tools that reset after a period of time can improve latency and cost in production settings. For example, if a tool result to fetch quarterly financial metrics from a database does not change until the next quarter, then caching can prevent repeated execution of that query for all users by saving the result and avoiding redundant database queries [57,58,159,160].

Associated Prompts

When attaching a set of individual, unrelated tools to a top-level supervisor or orchestration agent, the agent may not understand the nuances or complexities of how to use the tools. By appending tool-associated prompts to the supervisor's system prompt from an MCP server [1,25], the supervisor agent has more context about how to use the tools in a longer-running job, rather than the sole tool description and input parameter descriptions.

 doi:10.20944/preprints202512.1050.v1

Human-in-the-Loop

During tool execution, tools or agents may need human approval or clarification that pauses tool execution, either due to user or agent authentication or authorization, or the severity of the tool [32]. The human can either verify or confirm the tool execution, commonly with a button click or verbal approval [62–64]. The other approach is editing a tool call, where the human can manually change the input parameters to the tool from a UI, or with natural language back to the agent[32]. As stated previously, tools or agents may need human-in-the-loop approval due to being write-based tools, rather than read-only tools, which require further approval for production-grade applications.

Authorization

Authorization is a critical factor in the execution phase of agents, to prevent restricted access based on the user's role or status. If a tool or agent is selected for execution but the user lacks access, the run should stop altogether [40,41]. Authorization can be applied in both the retrieval and execution phases. In the retrieval phase, unauthorized tools or agents are filtered out before selection, so they never enter the final available set or DAG. In the execution phase, the system may retrieve and select from the full catalog (even if the user lacks access), but each tool or agent execution is checked and authorized, and the supervisor must replan or abort. The choice of where to enforce authorization depends on production environments. A common pattern is retrieval-time filtering for privacy and determinism, with execution-time checks giving the user transparency of the agent access [161].

4.4. Memory for Multi-Turn Tool Selection

Memory for tool selection for Large Language Model (LLM) agents enables the persistent accumulation of knowledge, sustained contextual awareness, and adaptive decision-making informed by historical interactions and prior experiences. Existing literature divides memory for AI systems into short-term and long-term memory, each serving distinct but complementary roles in multi-turn agentic interactions [162].

4.4.1. Short-Term Memory for Dynamic Tool Management

Short-term memory manages inputs within limited context windows, consisting of sensory inputs (e.g., tools, multimodal data) and working memory (session messages) [162]. The majority of flagship model context windows fall between 100,000 to 1,000,000 tokens [163,164], requiring effective strategies to prevent token overflow and prune context that distracts the model during task execution [165–168].

While performing tool and agent selection over multi-turn user interactions, removing and adding dynamic sets of tools and agents in the available context window of the system becomes critical. MemTool [18] introduces short-term dynamic tool memory that gives the LLM agent autonomy to prune its available dynamic tools and search for new ones across a session. Previous tool retrieval approaches focus on a single turn, rather than multi-turn tool selection [8,10,20,42].

4.4.2. Long-Term Memory for Personalized Tool Use

Long-term memory involves storing information for extended periods that persist across sessions, enabling the model to retain knowledge and learned tools over time [162,169,170]. It includes explicit memory (key facts, events) and implicit memory (skills, procedures), with explicit memory further dividing into episodic memory (personal experiences) and semantic memory (facts about the environment) [171].

Existing memory frameworks [172–175] manage the persistence of long-term interactions either through a vector database, knowledge graph, or traditional database, and contain built-in modules for context engineering [174,176]. Long-term tool memory approaches focus on personalized tool calling based on the user, adapting tool usage based on user preferences and historical interaction patterns [177–179].

 doi:10.20944/preprints202512.1050.v1

4.5. Evaluations

Evaluation spans two axes: (i) whether the system retrieves the right tools and (ii) whether execution produces correct task outcomes, with operational analytics guiding iteration.

4.5.1. Tool Retrieval Metrics

Tool retrieval is an information-retrieval problem where reporting standard IR metrics is common: Recall@k, Precision@k, MRR@k, nDCG@k, and mAP@k on ground-truth tool sets [180]. Other agent-specific tool retrieval metrics include tool completeness and correctness, or whether the expected tools were surfaced and called [181]. For tool-applied RAG retrieval [8], there is Context Precision and Context Recall to quantify ranking quality and coverage, consistent with common RAG evaluation workflows [182].

4.5.2. Execution Metrics

For execution-based metrics, LLM-as-a-judge is the established evaluation guideline for whether the LLM agent answered the question using the correct tools [12,144–147,150,183–186]. Task Completion indicates end-to-end success of agentic workflows and multi-round tool calling workflows [187].

4.5.3. Tool Analytics

Standard tool and agent analytics include metrics such as the total number of base AI units in the system and whether those base AI units span tools or agents. Operational telemetry metrics should cover latency of tools and agents, token usage, cost, error/failure rates, and popularity/performance by tool or agent [188]. Having predictive analytics on which tool and agent are used the most can also indicate poisoning techniques by MCP servers to influence retrieval and selection architectures [117].

5. Conclusions

Large Language Model (LLM) agents are evolving into complex systems capable of reasoning, acting, and collaborating through external tools, Model Context Protocol (MCP) servers, and other agents. As these systems scale, selecting the appropriate tool or agent efficiently and securely has become a key challenge for real-world deployment. This work has examined the landscape of tool and agent selection from both frontend and backend perspectives, outlining how production systems integrate retrieval, orchestration, execution, and memory to achieve scalable and reliable performance. We have also highlighted how emerging standards such as MCP and Agent-to-Agent (A2A) communication, together with mechanisms for authorization, human-in-the-loop verification, and context memory, are shaping the next generation of tool-using agents. Looking ahead, as agents become more autonomous, there are emerging retrieval systems not just for agents, tools, or MCP servers, but also skills, which are consolidated sets of instructions, tools, and agents that complete a task. Furthermore, as agents are fine-tuned for long-running tasks with upwards of 200–300 sequential tool calls, the aforementioned retrieval systems and context engineering techniques will be trained into the underlying model themselves. We hope that future research is guided by this modern work of agent and tool selection in production.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

References

1. Model Context Protocol. Tools Documentation (MCP Concepts). https://modelcontextprotocol.io/docs/concepts/tools, 2025.
2. OpenAI. Function Calling. https://platform.openai.com/docs/guides/function-calling?api-mode=chat, 2024. Accessed: 2025-08-06.
3. Anthropic. Tool Use with Claude. https://docs.anthropic.com/en/docs/agents-and-tools/tool-use/overview, 2024. Accessed: 2025-08-06.

 doi:10.20944/preprints202512.1050.v1

4. Microsoft Learn. Azure AI Agent Service (preview) — Overview, 2025. Accessed: 2025-11-08.
5. Amazon Web Services. Agents for Amazon Bedrock — Developer Guide, 2025. Accessed: 2025-11-08.
6. LlamaIndex. Function Calling with Agents (LlamaIndex Examples), 2025. Accessed: 2025-11-08.
7. deepset. Haystack Agents — Concepts and Usage, 2025. Accessed: 2025-11-08.
8. Lumer, E.; Subbiah, V.K.; Burke, J.A.; Basavaraju, P.H.; Huber, A. Toolshed: Scale Tool-Equipped Agents with Advanced RAG-Tool Fusion and Tool Knowledge Bases, 2024, [2410.14594]. Also published in ICAART 2025.
9. Lumer, E.; Gulati, A.; Subbiah, V.K.; Basavaraju, P.H.; Burke, J.A. ScaleMCP: Dynamic and Auto-Synchronizing Model Context Protocol Tools for LLM Agents. *arXiv preprint* **2025**, [2505.06416].
10. Qin, Y.; Liang, S.; Ye, Y.; et al. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-World APIs. *arXiv preprint* **2023**, [2307.16789].
11. Du, Y.; Wei, F.; Zhang, H. AnyTool: Self-Reflective, Hierarchical Agents for Large-Scale API Calls. *arXiv preprint* **2024**, [2402.04253].
12. Frank, K.; Gulati, A.; Lumer, E.; Campagna, S.; Subbiah, V.K. Jackal: A Real-World Execution-Based Benchmark Evaluating Large Language Models on Text-to-JQL Tasks, 2025, [2509.23579].
13. Aizawa, K.; Engineering, A. Writing Effective Tools for Agents — with Agents. https://www.anthropic.com/engineering/writing-tools-for-agents. Published September 11, 2025; accessed 2025-10-15.
14. Microsoft. AutoGen (AG2) — Multi-Agent Framework, 2025. Accessed: 2025-11-08.
15. Manus. Manus. https://manus.im/, 2025. Accessed: 2025-11-07.
16. CrewAI. CrewAI — Documentation, 2025. Accessed: 2025-11-08.
17. AgentScope Contributors. AgentScope: A Flexible Yet Robust Framework for Building AI Agents, 2024. Accessed: 2025-11-08.
18. Lumer, E.; Gulati, A.; Subbiah, V.K.; Basavaraju, P.H.; Burke, J.A. MemTool: Optimizing Short-Term Memory Management for Dynamic Tool Calling in LLM Agent Multi-Turn Conversations, 2025, [2507.21428].
19. Mo, G.; Zhong, W.; Chen, J.; Chen, X.; Lu, Y.; Lin, H.; He, B.; Han, X.; Sun, L. LiveMCPBench: Can Agents Navigate an Ocean of MCP Tools?, 2025, [arXiv:cs.AI/2508.01780].
20. Chen, Y.; Yoon, J.; Sachan, D.S.; et al. Re-Invoke: Tool Invocation Rewriting for Zero-Shot Tool Retrieval. *arXiv preprint* **2024**, [2408.01875].
21. Zheng, Y.; Li, P.; Liu, W.; et al. ToolRerank: Adaptive and Hierarchy-Aware Reranking for Tool Retrieval. *arXiv preprint* **2024**, [2403.06551].
22. Microsoft. GraphRAG — A Graph-based Approach to Retrieval-Augmented Generation, 2024. Accessed: 2025-11-08.
23. Sarmah, P.; et al. Hybrid RAG: Integrating Knowledge Graphs with Retrieval-Augmented Generation. *arXiv preprint arXiv:2408.XXXX* **2024**.
24. ToolBench Contributors. ToolBench: A Comprehensive Benchmark for Tool Learning, 2024. Accessed: 2025-11-08.
25. LangChain. Context Engineering for AI Agents with LangChain and Manus. https://www.youtube.com/watch?v=6_BcCthVvb8, 2025. YouTube video; presenters: Lance Martin (LangChain) and Yichao "Peak" Ji (Manus); accessed 2025-11-06.
26. AI, L. LangChain Documentation. https://docs.langchain.com/, 2025. Accessed: 2025-10-15.
27. Jones, A.; Kelly, C. Code execution with MCP: Building more efficient agents. https://www.anthropic.com/engineering/code-execution-with-mcp, 2025.
28. Anthropic. Introducing advanced tool use on the Claude Developer Platform. https://www.anthropic.com/engineering/advanced-tool-use, 2025. Engineering blog.
29. Vercel. AI Tools — Vercel AI SDK, 2025. Accessed: 2025-11-08.
30. LlamaIndex. AgentToolSpec API Reference (LlamaIndex), 2025. Accessed: 2025-11-08.
31. Amazon Web Services. Amazon GenAI — AgentCore (Serverless Agent Framework) Documentation, 2025. Accessed: 2025-11-08.
32. LangChain Blog. Introducing Ambient Agents. https://blog.langchain.com/introducing-ambient-agents/, 2024. Accessed: 2025-08-06.
33. OpenAI. Guardrails for Python (OpenAI), 2025. Accessed: 2025-11-08.
34. NVIDIA. NVIDIA NeMo Guardrails — Documentation, 2024. Accessed: 2025-11-08.
35. OWASP Foundation. OWASP Top 10 for Large Language Model Applications, 2024. Accessed: 2025-11-08.
36. NIST. NIST AI Safety Institute Consortium (AISIC), 2024. Accessed: 2025-11-08.

 doi:10.20944/preprints202512.1050.v1

37. Ouyang, S.; Yan, J.; Hsu, I.H.; Chen, Y.; Jiang, K.; Wang, Z.; Han, R.; Le, L.T.; Daruki, S.; Tang, X.; et al. ReasoningBank: Scaling Agent Self-Evolving with Reasoning Memory. *arXiv preprint* **2025**, [2509.25140].
38. Google. Function calling with the Gemini API. https://ai.google.dev/gemini-api/docs/function-calling, 2024. Accessed: 2025-08-06.
39. Gerganov, G.; contributors. llama.cpp: Port of Facebook's LLaMA model in C/C++, 2025. GitHub repository, accessed 2025-11-08.
40. Arcade. Tool Calling-Introduction. https://docs.arcade.dev/home/use-tools/tools-overview, 2024. Accessed: 2025-08-06.
41. Composio. Composio — Skills that evolve with your Agents, 2025. Website, accessed 2025-11-09.
42. Liu, M.M.; Garcia, D.; Parllaku, F.; Upadhyay, V.; Shah, S.F.A.; Roth, D. ToolScope: Enhancing LLM Agent Tool Use through Tool Merging and Context-Aware Filtering, 2025, [arXiv:cs.CL/2510.20036].
43. Microsoft Blog. Agents in AutoGen. https://microsoft.github.io/autogen/0.2/blog/2024/05/24/Agent/, 2024. Accessed: 2025-08-06.
44. Krishnan, N. AI Agents: Evolution, Architecture, and Real-World Applications, 2025, [arXiv:cs.AI/2503.12687].
45. Mistral AI. Agents & Conversations, 2025. Mistral Docs, accessed 2025-11-08.
46. Anthropic. Introducing Agent Skills. https://www.anthropic.com/news/skills, 2025. Accessed: 2025-10-17.
47. AI, L. LangGraph Agent Supervisor Tutorial. https://langchain-ai.github.io/langgraph/tutorials/multi_agent/agent_supervisor/, 2025. Accessed: 2025-10-15.
48. Microsoft. Handoff Orchestration in Semantic Kernel. https://learn.microsoft.com/en-us/semantic-kernel/frameworks/agent/agent-orchestration/handoff?pivots=programming-language-csharp, 2025. Accessed: 2025-10-15.
49. Li, M.; Zhao, Y.; Yu, B.; et al. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. *arXiv preprint* **2023**, [2304.08244].
50. Wang, Z.; Cheng, Z.; Zhu, H.; Fried, D.; Neubig, G. What Are Tools Anyway? A Survey from the Language Model Perspective, 2024, [arXiv:cs.CL/2403.15452].
51. Qu, C.; Dai, S.; Wei, X.; Cai, H.; Wang, S.; Yin, D.; Xu, J.; Wen, J.r. Tool Learning with Large Language Models: A Survey. *Frontiers of Computer Science* **2025**, *19*. https://doi.org/10.1007/s11704-024-40678-2.
52. Patil, S.G.; Zhang, T.; Wang, X.; Gonzalez, J.E. Gorilla: Large Language Model Connected with Massive APIs. *arXiv preprint* **2023**, [2305.15334].
53. Anthropic. Writing effective tools for AI agents — using AI agents, 2025. Engineering blog, accessed 2025-11-10.
54. Huang, T.; Jung, D.; Chen, M. Planning and Editing What You Retrieve for Enhanced Tool Learning. In Proceedings of the Findings of the Association for Computational Linguistics: NAACL 2024, 2024. Accessed 2025-11-10.
55. Ji, P. Context Engineering for AI Agents: Lessons from Building Manus, 2025. Manus Blog, accessed 2025-11-09.
56. LangChain. Middleware, 2025. Documentation, accessed 2025-11-09.
57. IETF. RFC 9111: HTTP Caching, 2022. Internet Standard, accessed 2025-11-09.
58. MDN Web Docs. Cache-Control header — HTTP, 2025. Documentation, accessed 2025-11-09.
59. OpenAI. Prompt caching. https://platform.openai.com/docs/guides/prompt-caching, 2025. Accessed: 2025-11-07.
60. Anthropic. Prompt caching with Claude. https://claude.com/blog/prompt-caching, 2025. Accessed: 2025-11-07.
61. Perplexity AI. Getting Started with Perplexity Tools. https://www.perplexity.ai/hub/getting-started, 2024. Accessed: 2025-08-06.
62. Anthropic. Claude Code Overview. https://docs.anthropic.com/en/docs/claude-code/overview, 2024. Accessed: 2025-08-06.
63. Cursor IDE. Tools. https://docs.cursor.com/agent/tools, 2024. Accessed: 2025-08-06.
64. OpenAI. Codex CLI. https://developers.openai.com/codex/cli/, 2025. Accessed: 2025-10-15.
65. AI, C. What is Cline? https://docs.cline.bot/getting-started/what-is-cline, 2025. Accessed: 2025-10-15.
66. GitHub. GitHub Copilot Features. https://docs.github.com/en/copilot/get-started/features, 2025. Accessed: 2025-10-15.
67. Continue Dev. Automating Documentation Updates with Continue CLI, 2025. Documentation, accessed 2025-11-08.

 doi:10.20944/preprints202512.1050.v1

68. FlowiseAI. Tutorials: Interacting with API Tools & MCP; Human In The Loop; Agentic RAG, 2025. Documentation, accessed 2025-11-08.
69. Open WebUI. Action Function, 2025. Documentation, accessed 2025-11-08.
70. LobeHub. Plugins, 2025. Documentation, accessed 2025-11-08.
71. GitHub Docs. Use GitHub Copilot Agents, 2024. Accessed: 2025-11-08.
72. OpenAI. What is the mentions feature for GPTs? https://help.openai.com/en/articles/8908924-what-is-the-mentions-feature-for-gpts, 2024. Accessed: 2025-08-06.
73. Windsurf AI. Chat Overview. https://docs.windsurf.com/chat/overview, 2024. Accessed: 2025-08-06.
74. Anthropic. Customize Claude Code with plugins. https://www.anthropic.com/news/claude-code-plugins, 2025. Accessed: 2025-10-17.
75. Windsurf. Windsurf — Slash Commands Reference, 2025. Accessed: 2025-11-08.
76. Mistral AI. Function Calling. https://docs.mistral.ai/capabilities/function_calling/, 2024. Accessed: 2025-08-06.
77. Meta. Tool Calling. https://llama.developer.meta.com/docs/features/tool-calling, 2024. Accessed: 2025-08-06.
78. xAI. Function Calling. https://docs.x.ai/docs/guides/function-calling, 2024. Accessed: 2025-08-06.
79. Groq. Compound. https://console.groq.com/docs/agentic-tooling#use-cases, 2024. Accessed: 2025-08-06.
80. Google Cloud. Vertex AI Generative AI — Introduction to function calling. https://cloud.google.com/vertex-ai/generative-ai/docs/multimodal/function-calling, 2024. Accessed: 2025-08-06.
81. Amazon Web Services. Use a tool to complete an Amazon Bedrock model response. https://docs.aws.amazon.com/bedrock/latest/userguide/tool-use.html, 2024. Accessed: 2025-08-06.
82. Google Gemini CLI. Gemini CLI Core: Tools API. https://github.com/google-gemini/gemini-cli/blob/main/docs/core/tools-api.md, 2024. Accessed: 2025-08-06.
83. IBM. Tool Calling. https://www.ibm.com/watsonx/developer/capabilities/tool-calling, 2024. Accessed: 2025-08-06.
84. Reka AI. Function Calling. https://docs.reka.ai/chat/function-calling, 2024. Accessed: 2025-08-06.
85. Inflection AI. Inflection Inference API (1.0.0). https://developers.inflection.ai/api/docs, 2024. Accessed: 2025-08-06.
86. SmolAgents. SmolAgents Tools. https://huggingface.co/docs/smolagents/tutorials/tools, 2024. Accessed: 2025-08-06.
87. Mistral AI. Function Calling, 2025. Mistral Docs, accessed 2025-11-08.
88. Ollama. How to Create Tools to Extend AI with Functions, 2024. Blog post, accessed 2025-11-08.
89. Radford, A.; Kim, J.W.; Xu, T.; Brockman, G.; McLeavey, C.; Sutskever, I. Robust Speech Recognition via Large-Scale Weak Supervision. *arXiv preprint arXiv:2212.04356* **2022**.
90. Baevski, A.; Zhou, H.; Mohamed, A.; Auli, M. wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations. *arXiv preprint arXiv:2006.11477* **2020**.
91. Zhang, Y.; et al. USM: Scaling Automatic Speech Recognition Beyond 100 Languages. *arXiv preprint arXiv:2303.01037* **2023**.
92. OpenAI. Realtime API Overview. https://platform.openai.com/docs/guides/realtime, 2024. Accessed: Oct. 7, 2025.
93. Sesame AI. Crossing the Uncanny Valley of Conversational Voice. https://www.sesame.com/research/crossing_the_uncanny_valley_of_voice, 2024. Accessed: 2025-08-06.
94. Cohere. Basic usage of tool use (function calling). https://docs.cohere.com/docs/tool-use-overview, 2024. Accessed: 2025-08-06.
95. Liu, H.; Li, C.; Wu, Q.; Lee, Y.J. Visual Instruction Tuning. *arXiv preprint arXiv:2304.08485* **2023**.
96. Lumer, E.; Cardenas, A.; Melich, M.; Mason, M.; Dieter, S.; Subbiah, V.K.; Basavaraju, P.H.; Hernandez, R. Comparison of Text-Based and Image-Based Retrieval in Multimodal Retrieval Augmented Generation Large Language Model Systems, 2025, [arXiv:cs.CL/2511.16654].
97. Microsoft. Visual Studio Code: Intelligent Code Assistance. https://code.visualstudio.com/docs/editor/intellisense, 2024. Accessed: 2025-08-06.
98. Park, J.S.; O'Brien, J.C.; Cai, C.J.; Morris, M.R.; Liang, P.; Bernstein, M.S. Generative Agents: Interactive Simulacra of Human Behavior. *arXiv preprint arXiv:2304.03442* **2023**.
99. Mem0. Mem0: The Memory Layer for Personalized AI. https://mem0.ai/, 2025. Accessed: 2025-08-06.
100. LlamaIndex. Long-Term Memory in LlamaIndex, 2025. Accessed: 2025-11-08.
101. LangChain AI. LangGraph — Persistence and Checkpointing, 2025. Accessed: 2025-11-08.

 doi:10.20944/preprints202512.1050.v1

102. Python Software Foundation. Python Documentation. https://docs.python.org/3/, 2025. Accessed: 2025-10-07.
103. Microsoft. TypeScript Documentation. https://www.typescriptlang.org/docs/, 2025. Accessed: 2025-10-07.
104. Google DeepMind and Google Research. Announcing the Agent2Agent Protocol (A2A). https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/, 2025. Accessed: 2025-10-07.
105. IBM Research. The simplest protocol for AI agents to work together. https://research.ibm.com/blog/agent-communication-protocol-ai, 2025. Accessed: 2025-10-07.
106. Heule, S.; Jia, E.; Jain, N. Improving agent with semantic search. https://cursor.com/blog/semsearch, 2025. Cursor research blog post.
107. Huang, N. How agents can use filesystems for context engineering. https://blog.langchain.com/how-agents-can-use-filesystems-for-context-engineering/, 2025. LangChain blog post.
108. Gao, Y.; Xiong, Y.; et al. Retrieval-Augmented Generation for Large Language Models: A Survey. *arXiv preprint* **2024**, [2312.10997].
109. Peng, B.; Zhu, Y.; Liu, Y.; et al. Graph Retrieval-Augmented Generation: A Survey. *arXiv preprint* **2024**, [2408.08921].
110. Wu, M.; Zhu, T.; Han, H.; et al. SEAL-Tools: Self-Instruct Tool Learning Dataset for Agent Tuning and Detailed Benchmark. *arXiv preprint* **2024**, [2405.08355].
111. Han, H.; Shomer, H.; Wang, Y.; Lei, Y.; Guo, K.; Hua, Z.; Long, B.; Liu, H.; Tang, J. RAG vs. GraphRAG: A Systematic Evaluation and Key Insights, 2025, [arXiv:cs.IR/2502.11371].
112. Lumer, E.; Basavaraju, P.H.; Mason, M.; Burke, J.A.; Subbiah, V.K. Graph RAG-Tool Fusion, 2025, [2502.07223].
113. Sarmah, B.; Hall, B.; Rao, R.; Patel, S.; Pasquali, S.; Mehta, D. HybridRAG: Integrating Knowledge Graphs and Vector Retrieval Augmented Generation for Efficient Information Extraction, 2024, [arXiv:cs.CL/2408.04948].
114. OpenAI. Embedding Models. https://platform.openai.com/docs/guides/embeddings/embedding-models, 2024. Accessed: 2025-10-08.
115. Cohere. Cohere Embed Models. https://docs.cohere.com/docs/cohere-embed, 2024. Accessed: 2025-10-08.
116. Google Cloud. Vertex AI Embeddings (preview). https://cloud.google.com/vertex-ai/generative-ai/docs/embeddings, 2025. Accessed: 2025-10-08.
117. Shi, J.; Yuan, Z.; Tie, G.; Zhou, P.; Gong, N.Z.; Sun, L. Prompt Injection Attack to Tool Selection in LLM Agents, 2025, [arXiv:cs.CR/2504.19793].
118. Trivedi, H.; Balasubramanian, N.; Khot, T.; Sabharwal, A. Interleaving Retrieval with Chain-of-Thought Reasoning for Knowledge-Intensive Multi-Step Questions. In Proceedings of the NeurIPS, 2023.
119. Lumer, E.; Nizar, F.; Gulati, A.; Basavaraju, P.H.; Subbiah, V.K. Tool-to-Agent Retrieval: Bridging Tools and Agents for Scalable LLM Multi-Agent Systems, 2025, [arXiv:cs.CL/2511.01854].
120. Nizar, F.; Lumer, E.; Gulati, A.; Basavaraju, P.H.; Subbiah, V.K. Agent-as-a-Graph: Knowledge Graph-Based Tool and Agent Retrieval for LLM Multi-Agent Systems, 2025, [arXiv:cs.CL/2511.18194].
121. Singh, A.; Ehtesham, A.; Kumar, S.; Talaei Khoei, T. Agentic Retrieval-Augmented Generation: A Survey on Agentic RAG. *arXiv preprint arXiv:2501.09136* **2025**.
122. Maragheh, R.Y.; Vadla, P.; Gupta, P.; Zhao, K.; Inan, A.; Yao, K.; Xu, J.; Kanumala, P.; Cho, J.; Kumar, S. ARAG: Agentic Retrieval Augmented Generation for Personalized Recommendation. *arXiv preprint arXiv:2506.21931* **2025**.
123. Jin, B.; Zeng, H.; Yue, Z.; Yoon, J.; Arik, S.Ö.; Wang, D.; Zamani, H.; Han, J. Search-R1: Training LLMs to Reason and Leverage Search Engines with Reinforcement Learning. *arXiv preprint arXiv:2503.09516* **2025**.
124. Lumer, E.; Melich, M.; Zino, O.; Kim, E.; Dieter, S.; Basavaraju, P.H.; Subbiah, V.K.; Burke, J.A.; Hernandez, R. Rethinking Retrieval: From Traditional Retrieval Augmented Generation to Agentic and Non-Vector Reasoning Systems in the Financial Domain for Large Language Models, 2025, [arXiv:cs.CL/2511.18177].
125. Wang, X.; Chen, Y.; Yuan, L.; Zhang, Y.; Li, Y.; Peng, H.; Ji, H. Executable Code Actions Elicit Better LLM Agents, 2024, [arXiv:cs.CL/2402.01030].
126. Varda, K.; Pai, S. Code Mode: the better way to use MCP. https://blog.cloudflare.com/code-mode/, 2025. Accessed: 2025-11-08.
127. Hacker News. Discussion: Code Mode — the better way to use MCP. https://news.ycombinator.com/item?id=45830318, 2025. Accessed: 2025-11-08.
128. Mistral AI. Tools, 2025. Mistral Docs, accessed 2025-11-08.
129. Moonshot AI. Kimi K2 — Thinking. https://moonshotai.github.io/Kimi-K2/thinking.html, 2025. Accessed: 2025-11-07.

 doi:10.20944/preprints202512.1050.v1

130. Anthropic. Use XML tags to structure your prompts. https://anthropic.mintlify.app/en/docs/build-with-claude/prompt-engineering/use-xml-tags, 2025. Accessed: 2025-11-07.
131. OpenAI. Reasoning models. https://platform.openai.com/docs/guides/reasoning, 2025. Accessed: 2025-11-07.
132. Together AI. DeepSeek-R1 Quickstart. https://docs.together.ai/docs/deepseek-r1, 2025. Accessed: 2025-11-07.
133. veRL Maintainers. Search Tool Integration — Multi-Turn RL. https://verl.readthedocs.io/en/latest/sglang_multiturn/search_tool_example.html, 2025. Accessed: 2025-11-07.
134. Moonshot AI. Kimi K2: Open Agentic Intelligence. *arXiv preprint arXiv:2507.20534* **2025**.
135. Google AI for Developers. Context caching — Gemini API. https://ai.google.dev/gemini-api/docs/caching?lang=py, 2025. Accessed: 2025-11-07.
136. Cursor Team. Introducing Cursor 2.0 and Composer · Cursor. https://cursor.com/blog/2-0, 2025. Accessed: October 2025.
137. LangChain. Deep Agents overview — Docs by LangChain. https://docs.langchain.com/oss/python/deepagents/overview, 2025. Accessed: 2025-11-08.
138. OpenDevin Contributors. OpenDevin — Autonomous Software Engineering Agent, 2024. Accessed: 2025-11-08.
139. Anthropic. Introducing Claude Sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, 2025. Accessed: 2025-10-17.
140. Qian, C.; Acikgoz, E.C.; He, Q.; Wang, H.; Chen, X.; Hakkani-Tür, D.; Tur, G.; Ji, H. ToolRL: Reward is All Tool Learning Needs. *arXiv preprint arXiv:2504.13958* **2025**.
141. Dong, G.; Chen, Y.; Li, X.; Jin, J.; Qian, H.; Zhu, Y.; Mao, H.; Zhou, G.; Dou, Z.; Wen, J.R. Tool-Star: Empowering LLM-Brained Multi-Tool Reasoner via Reinforcement Learning. *arXiv preprint arXiv:2505.16410* **2025**.
142. Anonymous. Reinforcement Learning with Verifiable Rewards: GRPO's Effective Loss. *arXiv preprint arXiv:2503.06639* **2025**.
143. Singh, J.; Magazine, R.; Pandya, Y.; Nambi, A. Agentic Reasoning and Tool Integration for LLMs via Reinforcement Learning, 2025, [arXiv:cs.AI/2505.01441].
144. Yao, S.; Shinn, N.; Razavi, P.; Narasimhan, K. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. *arXiv preprint arXiv:2406.12045* **2024**.
145. Barres, V.; Dong, H.; Ray, S.; Si, X.; Narasimhan, K. τ^2-Bench: Evaluating Conversational Agents in a Dual-Control Environment. *arXiv preprint arXiv:2506.07982* **2025**.
146. Chen, C.; Hao, X.; Liu, W.; Huang, X.; Zeng, X.; Yu, S.; Li, D.; Wang, S.; Gan, W.; Huang, Y.; et al. ACEBench: Who Wins the Match Point in Tool Usage? *arXiv preprint arXiv:2501.12851* **2025**.
147. Zhong, L.; Du, Z.; Zhang, X.; Hu, H.; Tang, J. ComplexFuncBench: Exploring Multi-Step and Constrained Function Calling under Long-Context Scenario. *arXiv preprint arXiv:2501.10132* **2025**.
148. Wang, X.; Wang, Z.; Liu, J.; Chen, Y.; Yuan, L.; Peng, H.; Ji, H. MINT: Evaluating LLMs in Multi-turn Interaction with Tools and Language Feedback. *arXiv preprint arXiv:2309.10691* **2024**.
149. Wang, P.; Wu, Y.; Wang, Z.; Liu, J.; Song, X.; Peng, Z.; Deng, K.; Zhang, C.; Wang, J.; Peng, J.; et al. MTU-Bench: A Multi-granularity Tool-Use Benchmark for Large Language Models. In Proceedings of the International Conference on Learning Representations (ICLR), 2025. arXiv:2410.11710.
150. Patil, S.G.; Mao, H.; Yan, F.; Ji, C.C.; Suresh, V.; Stoica, I.; Gonzalez, J.E. The Berkeley Function Calling Leaderboard (BFCL): From Tool Use to Agentic Evaluation of Large Language Models. In Proceedings of the Proceedings of the 42nd International Conference on Machine Learning (ICML), 2025.
151. WebArena Contributors. WebArena: Open, Reproducible Web Environments for Agents, 2024. Accessed: 2025-11-08.
152. VisualWebArena Contributors. VisualWebArena: A Visual Web Navigation Benchmark for Multimodal Agents, 2024. Accessed: 2025-11-08.
153. BrowserGym Contributors. BrowserGym: A Browser Automation and Web Agent Benchmark, 2024. Accessed: 2025-11-08.
154. AgentBoard Contributors. AgentBoard: A Unified Platform for Evaluating LLM Agents, 2024. Accessed: 2025-11-08.
155. Schmid, P. The New Skill in AI is Not Prompting, It's Context Engineering, 2025. Blog post, published 2025-06-30, accessed 2025-11-10.
156. LangChain. Context engineering in agents, 2025. Documentation, accessed 2025-11-09.

 doi:10.20944/preprints202512.1050.v1

157. Martin, L. Context Engineering for Agents, 2025. Blog post, accessed 2025-11-09.
158. OpenAI. Structured model outputs, 2025. Documentation, accessed 2025-11-09.
159. Redis. Redis, 2025.
160. LangChain. Node-level caching in LangGraph, 2025. Changelog, accessed 2025-11-10.
161. Databricks. Agent Framework — Databricks GenAI, 2025. Accessed: 2025-11-08.
162. Wu, S.; Others. Human Memory for AI Memory: Building Agent Memory from Cognitive Psychology, 2025. *arXiv preprint*.
163. Vellum. LLM Context Window Comparison, 2025.
164. Shang, S. LongRoPE 2: Nearly Lossless LLM Context Extension, 2025. *arXiv preprint*.
165. Laban, S. LLMs Lost in Multi-Turn Conversation, 2025. *arXiv preprint*.
166. Hong, S. Context Management for Multi-Turn Conversations, 2025. *arXiv preprint*.
167. Chirkova, S. Provence: Efficient and Robust Context Management, 2025. *arXiv preprint*.
168. Schmid, S. Context Engineering for LLM Agents, 2025. *arXiv preprint*.
169. Park, J.S.; Others. Generative Agents: Interactive Simulacra of Human Behavior, 2023. *arXiv preprint arXiv:2304.03442*.
170. Zhong, W.; Others. MemoryBank: Enhancing Large Language Models with Long-Term Memory, 2023. *arXiv preprint*.
171. Alake, R. Agent Memory: Episodic and Semantic Memory for LLMs, 2025.
172. Mem0. Mem0: The Memory Layer for AI Applications, 2025.
173. Zep. Zep: Long-Term Memory for AI Assistants, 2025.
174. Letta. Letta: Build Stateful LLM Applications, 2025.
175. Packer, C.; Others. MemGPT: Towards LLMs as Operating Systems, 2024. *arXiv preprint arXiv:2310.08560*.
176. Xu, S. AMem: Agentic Memory for LLM Agents, 2025. *arXiv preprint*.
177. Zhang, W.; Zhang, X.; Zhang, C.; Yang, L.; Shang, J.; Wei, Z.; Zou, H.P.; Huang, Z.; Wang, Z.; Gao, Y.; et al. PersonaAgent: When Large Language Model Agents Meet Personalization at Test Time. https://arxiv.org/abs/2506.06254, 2025. Accessed: 2025-10-17.
178. Hao, Y.; Cao, P.; Jin, Z.; Liao, H.; Chen, Y.; Liu, K.; Zhao, J. Evaluating Personalized Tool-Augmented LLMs from the Perspectives of Personalization and Proactivity. https://arxiv.org/abs/2503.00771, 2025. Accessed: 2025-10-17.
179. Cheng, Z.; Wang, H.; Liu, Z.; Guo, Y.; Guo, Y.; Wang, Y.; Wang, H. ToolSpectrum: Towards Personalized Tool Utilization for Large Language Models. https://arxiv.org/abs/2505.13176, 2025. Accessed: 2025-10-17.
180. Manning, C.D.; Raghavan, P.; Schütze, H. *Introduction to Information Retrieval*; Cambridge University Press: Cambridge, UK, 2008.
181. DeepEval. Tool Correctness. https://deepeval.com/docs/metrics-tool-correctness, 2025. Accessed: 2025-10-17.
182. Ragas. Metrics Overview. https://docs.ragas.io/en/latest/concepts/metrics/, 2025. Accessed: 2025-10-17.
183. OpenAI. Evaluation Best Practices. https://platform.openai.com/docs/guides/evaluation-best-practices, 2025. Accessed: 2025-10-17.
184. OpenAI. Working with evals. https://platform.openai.com/docs/guides/evals, 2025. Accessed: 2025-10-17.
185. Ragas. Response Relevancy. https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/answer_relevance/, 2025. Accessed: 2025-10-17.
186. Ragas. Ragas Metric Faithfulness. https://docs.ragas.io/en/latest/concepts/metrics/available_metrics/faithfulness/, 2025. Accessed: 2025-10-17.
187. DeepEval. Task Completion. https://deepeval.com/docs/metrics-task-completion, 2025. Accessed: 2025-10-17.
188. LangChain. Monitor projects with dashboards. https://docs.langchain.com/langsmith/dashboards, 2025. Accessed: 2025-10-17.