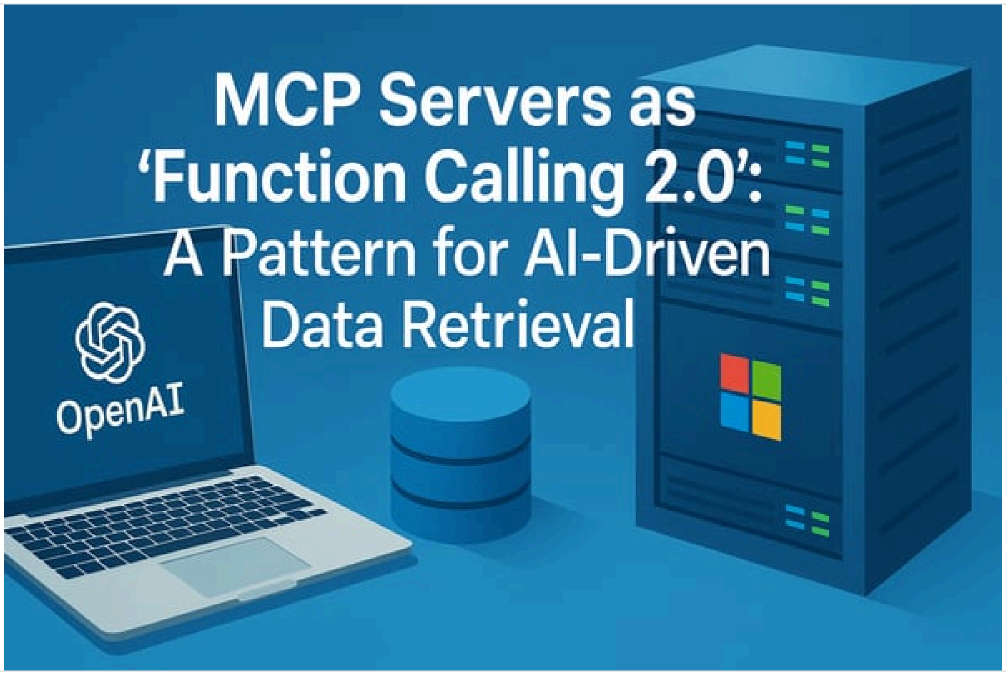
# The Model Context Protocol (MCP) for AI Tool Integration

Published August 3, 2025 65 min read



## MCP Servers as "Function Calling 2.0": A Pattern for AI-Driven Data Retrieval

Modern AI systems are increasingly expected to **retrieve live data, execute actions, and incorporate external knowledge** seamlessly. Traditional function calling features (as seen in many LLM APIs) mark an important first step, but they remain limited to one-off calls to predefined functions. Enter the **Model Context Protocol (MCP)** – an emerging open standard that takes function integration to the next level. MCP servers act as **universal connectors** between AI models and the world of data, earning the nickname "Function Calling 2.0" for their ability to orchestrate richer, ongoing tool interactions. This report provides a deep dive into MCP servers: their architecture, evolution from basic function calls, improvements over prior approaches, key use cases, comparisons with alternatives (GraphQL, LangChain, RAG, etc.), and technical insights into their operation. It is structured for engineers, system architects, and AI researchers seeking an in-depth understanding of this new pattern.

## 1. What Are MCP Servers? – Definition and Architectural Role

**Model Context Protocol (MCP)** is an open standard (introduced by Anthropic in late 2024) that **standardizes how AI assistants connect to external data sources and tools**. In an MCP architecture, an AI application (the *MCP host*, e.g. ChatGPT, Claude, a coding IDE assistant, etc.) establishes a **persistent two-way connection** to one or more *MCP servers* 12†L131-L139** . Each MCP server is a program (which can run locally or remotely) that **exposes a set of capabilities – "tools," "resources," and "prompts" – to the AI**. The AI application runs an *MCP client* component for each server, handling communication. In essence, **MCP servers are modular connectors that provide the AI with access to external functionalities or data, via a uniform protocol**.

*Fig. 1: An example MCP setup – An AI Assistant (LLM-based application with an MCP client) connected to multiple MCP servers (e.g., one for Slack messages, one for a company database, one for a local filesystem). The connections use JSON-RPC over various transports (stdio for local, SSE or WebSocket for remote), enabling* ***stateful, two-way communication*** *between the AI and each data source.*

At a high level, you can think of MCP as a **"USB-C port for AI applications"** – a standardized interface through which an AI can plug into any tool or data source that has an MCP server. This uniformity replaces the fragmented, bespoke integrations of the past. Analogy: without MCP, integrating each AI system with each external service was like having *N×M custom adapters* – e.g. one custom API for GPT-4 to access your database, another for Claude to access the same DB, etc. In fact, *"before MCP"* one needed separate connectors or plugins for every combination of AI and service, a patchwork approach where *each model had its own format and translation layer for the same task*. *"After MCP,"* by contrast, we have **one standard protocol** – a single "menu" and service protocol all AI models can use to interact with any tool. This dramatically reduces the integration complexity: *"instead of maintaining separate connectors for each data source, developers can now build against a standard protocol"*, paving the way for **scalable, context-aware AI architectures**.

**Architecturally, an MCP server plays the role of an external tool provider in a client–server model.** The AI (client) and server maintain a persistent session, typically over JSON-RPC 2.0 messaging. The *MCP data layer* defines standard methods and data structures for the server to advertise and provide:

- **Tools:** Executable functions or operations the AI can invoke (analogous to function calls) – e.g. "queryDatabase", "sendEmail", "openBrowserTab". Tools may have side effects and are described with input/output schemas and descriptions.
- **Resources:** Read-only data sources that the AI can query for information – e.g. a vector database of documents, a filesystem, a knowledge base or search index. Resources provide context **similar to Retrieval-Augmented Generation (RAG)** systems, but via a standardized interface (for example, a `searchArticles` resource might return relevant documents for a query).
- **Prompts:** Reusable prompt templates or contextual hints that the server can supply to help the AI formulate requests or interpret outputs – e.g. a template for an SQL query or an example workflow. These help guide the AI's behavior in complex interactions.

Each of these primitives has standard discovery and use methods (e.g. `tools/list`, `tools/call`, `resources/list`, `resources/read`, etc.) so the AI can list what a server offers and then utilize them. Crucially, **MCP communication is two-way** – not only can the AI call the server's tools, but the server can also send **notifications or requests back**. For instance, the server might push a notification that new data is available, or even request the AI to produce a completion (via a *"sampling"* primitive) if the server needs help from the model to, say, summarize text or compose a query. This bidirectional design enables dynamic, ongoing exchanges of context rather than a single request/response. The MCP server thus becomes an active participant in the AI's reasoning loop, not just a passive API.

From a systems perspective, **MCP servers serve as secure brokers between AI and data**. They often incorporate **access control, authentication, and filtering**, since they expose potentially sensitive tools/data. The MCP spec supports multiple transport layers (from local STDIO pipes for on-device connectors to remote HTTP+SSE streams with token auth for cloud servers) (Source: modelcontextprotocol.io). But the *data exchange format* remains JSON-RPC for consistency across transports (Source: modelcontextprotocol.io)(Source: modelcontextprotocol.io). This means any MCP-compliant client can talk to any MCP server. In summary, MCP servers occupy a **crucial architectural role**: they are the **standard adapters that plug AI into the wider digital ecosystem**, enabling models to maintain context across disparate systems and take actions in the real world in a controlled, interoperable manner.

## 2. From Traditional APIs to OpenAI Function Calling – Evolution of "Function Calling"

To appreciate MCP as "function calling 2.0," it's useful to recap how AI function integration has evolved:

- **Traditional API Calls (pre-LLM or early LLM era):** If an AI system needed external data (e.g. current weather or a database lookup), the typical approach was completely outside the model – e.g. a developer would capture the user's intent and manually invoke an API, then feed the result back into the model prompt. There was no direct model-initiated calling. Each integration was custom-coded, and the model itself had no native notion of a "function call."

- **OpenAI Plugins (early 2023):** A step toward generalizing tool use, ChatGPT plugins provided an API schema that the model could call via specially formatted responses. Still, plugins were largely specific to ChatGPT and required the model to generate an exact API call format in text.

- **LLM Function Calling (mid 2023 onward):** OpenAI's introduction of *native function calling* in models like GPT-4 allowed developers to **define functions that the model could call** by name, with arguments in JSON, all within the model's output format. For example, one could define a function `getWeather(location, unit)`; if the user asks for weather info, the model can respond with a JSON like `{"name": "getWeather", "arguments": {"location": "Chennai", "unit": "Celsius"}}`. The calling client (OpenAI's API or SDK) would detect this and execute the actual function, then return the result to the model to incorporate into the final answer.

This was a significant breakthrough: **LLMs could "decide" to use tools** mid-response. Each major provider implemented a variant (OpenAI, Anthropic's Claude, Google's PaLM/Gemini, Meta's Llama all have similar constructs). The common pattern:

1. *Function definitions:* Developers register what functions (or tool APIs) the model has access to, including their parameters and purpose.

2. *Model decides on tool use:* When a user prompt comes in, the LLM's logic determines if any of those functions is needed to fulfill the request.

3. *Structured call output:* Instead of a final answer, the model outputs a **structured function call request** (usually JSON) indicating the function name and arguments.

4. *Execution and continuation:* The system executes the function outside the model, gets the result (e.g. the weather data), and feeds it back to the model. The model then continues, now armed with fresh data, to produce a complete answer for the user.

This made AI outputs more accurate and dynamic – no more "I'm sorry, I cannot access that info" if a function was available. However, **function calling 1.0 has limitations**:

- It's **limited to predefined functions**. The model can only call what the developer explicitly listed. There's *"no universal standard yet"* for function schemas across models – each vendor had its own JSON format, meaning functions had to be defined separately for each model family.

- **One-shot calls**: A typical function call workflow is *single-turn*. The model triggers a call, getsresult, then moves on. The model doesn't *maintain a rich dialog* with the tool beyond that call. It's essentially a remote procedure call (RPC) injected into the conversation.

- **No built-in context memory across calls**: The model doesn't inherently remember previous function outputs unless they were included in the conversation history. The function integration itself is stateless – each call is independent.

- **Fragmentation and scaling issues**: As the number of possible functions grows, managing them and expecting the model to choose correctly becomes challenging. Also integrating the same functions into multiple AI systems (without a standard) meant duplicate effort.

In summary, the advent of function calling was a big leap, letting LLMs perform actions like API calls. Yet, it was still a somewhat **manual, siloed approach** – bound to each vendor's API and not inherently designed for multi-step or cross-system orchestration. Developers began to crave a more robust, standardized way to connect AI to the wide world of tools and data.

## 3. MCP Servers – A Next-Generation Framework for Function Calling

**MCP can be viewed as "function calling 2.0" because it generalizes and extends the concept of LLM-initiated tool use in several important ways:**

- **Standardization across platforms:** MCP is an *open protocol*, not tied to a single model or vendor. It provides a universal language for describing tools and data. Once a data source has an MCP server, *any compliant AI assistant can use it*. This is akin to how USB standardized device connections. For function calling, this is a game changer: instead of each LLM having custom functions, they all can interface with the same MCP server for, say, a database, via the same JSON-RPC calls. This interoperability dramatically reduces the integration burden (the *N×M* problem).

- **Rich, **contextual interactions (beyond one-shot):** MCP is designed for **multi-turn, dynamic tool usage**. An AI can list available actions, ask for data, call a tool, get a result, then decide on another call – all within one continuous session. The MCP session is *stateful*, meaning the server can carry context from one call to the next. For example, if an MCP server returns page 1 of search results in one call, a subsequent call for "page 2" can be interpreted in context (no need to repeat the query). The protocol allows maintaining *session state and memory* on the server side (more on that in Section 8), enabling workflows that span multiple steps. This addresses scenarios where one function call isn't enough.

- **Two-way communication & continuous context:** Unlike basic function calls, MCP servers can proactively send information or request input. Through **notifications**, an MCP server might stream intermediate results or alert the AI client of an update (e.g. "file upload 50% complete") (Source: modelcontextprotocol.io). Servers can also query the AI (via the *sampling* primitive) or the user (via *elicitation*) if needed. This two-way channel transforms the interaction model: *MCP is more like an ongoing conversation with a tool*, rather than a single API call. It lets AI assistants receive *dynamic context updates in real-time*, making them far more responsive to changing data.

- **Unified interface for diverse operations:** MCP servers don't just handle "functions" in the narrow sense. They can provide data resources and prompt templates too. This means an MCP server can supply an AI with a **bundle of contextual support**: for example, a server for a knowledge base might provide a *resource* listing relevant document texts (like a RAG retrieval), *tools* for advanced queries or updates, and *prompts* with suggested follow-up questions. All of this is delivered in a cohesive way. *MCP thus goes beyond function calling to also standardize data retrieval (read operations) and even prompt patterns.* In effect, it subsumes function calling and situates it in a broader context mechanism.

- **Scalability and reuse:** By decoupling tool implementations into MCP servers, the approach becomes more scalable and maintainable. One MCP server can serve many AI clients (e.g., a single enterprise MCP server for a CRM database could be used by a customer support chatbot, a sales assistant bot, and a data analytics assistant, all via the same interface). This *reduces duplication* and centralizes maintenance. As F22 Labs noted, function calling can become complex as functions proliferate, whereas MCP is *"designed to handle complex integrations at scale"*. New tools added to a server are immediately available to any AI that connects, without code changes on the AI side beyond initial capability negotiation.

- **Maintaining long-term context and memory:** MCP facilitates **persistent context** much better than vanilla function calls. Since the connection is persistent and the server can store state, an AI agent can have a form of "working memory" via the MCP server. For example, an MCP server connected to a code editor could keep track of which files have been opened or what the current cursor position is, across multiple queries. This enables more coherent multi-step assistance. In fact, MCP is explicitly touted for use cases requiring *"reasoning across multiple exchanges"* and *"maintaining long-term memory for coherent user experiences"*. (Function calling alone doesn't manage multi-turn state – that burden fell on external agent frameworks.)

- **Secure and controlled tool use:** With function calling, one had to bake in any necessary permission or safety checks in the function implementation. MCP adds an extra layer of standardized control – tools can be exposed or filtered in a consistent manner, and servers define *"capabilities"* during the initial handshake. The protocol emphasizes secure data handling and standardized access control (e.g., servers declare what they can do and clients only allow what's necessary). This structure makes it easier to reason about security and audit what an AI is allowed to do, compared to ad-hoc function hooking.

In short, **MCP servers elevate function calling into a full-fledged **framework for AI-tool interoperability**. They preserve the strengths of function calling (structured, machine-readable operations beyond the LLM's inherent knowledge) and **add flexibility, context management, and standardization**. It's not just an API call – it's an ongoing *protocol*.

**Improvements in action:** Suppose a user asks an AI assistant, *"Find any critical customer tickets about payment issues in the last month and create a summary report."* With classic function calls, you might have a `searchTickets` function and a `summarize` function, and the model might call them one by one, possibly requiring the developer to orchestrate the multi-step logic. With MCP, the assistant can connect to a **Helpdesk MCP server** which provides a *resource* for ticket search and a *tool* for creating a report entry. The AI can iteratively use the resource (which might handle pagination, etc.), and then call the report tool, all within one session. The server might even provide a prompt template for formatting the summary. The context (e.g. the found tickets) can be kept on the server so the AI doesn't need to stuff everything into the prompt at once, avoiding token overhead. The result is a more fluid and intelligent exchange between the AI and the external system. This kind of **agentic workflow** is precisely what MCP is designed for.

## 4. Key Use Cases and Industry Applications of the MCP Pattern

By enabling AI-driven data retrieval and action across many systems, MCP servers unlock advanced capabilities in various domains. Some **key use cases and industries** benefiting from the MCP pattern include:

- **Customer Support and Service Automation:** AI assistants in customer support can use MCP to pull in relevant customer data, knowledge base articles, and even perform ticket actions. For example, a support chatbot could connect to a **Knowledge Base MCP server** (to retrieve solution articles) and a **CRM MCP server** (to fetch a user's order history or account status) during a conversation. This yields context-aware answers instead of generic responses. *IT support and helpdesk knowledge management* are cited as prime applications – e.g. using semantic search on past tickets to suggest solutions, or auto-routing issues to the right team based on content (Source: byteplus.com). An MCP-driven FAQ bot can deliver **precise, context-aware responses** by tapping into up-to-date internal documentation (Source: byteplus.com). Companies are already exploring this: Anthropic mentions "Claude for Work" using MCP connectors to internal systems for enterprise support scenarios.

- **Knowledge Management and Enterprise Search:** Organizations often have vast stores of unstructured data (documents, wikis, drive files, emails). MCP servers provide a way for AI to **search and retrieve enterprise knowledge** securely. For instance, there are MCP servers for Google Drive, Slack, Confluence, databases like Postgres, etc.. By connecting an AI assistant to these, one can ask natural language questions and have the AI fetch answers from across the company's data silos. This is essentially **retrieval-augmented generation (RAG)** on steroids – instead of a custom pipeline, the AI uses a standard set of "search" and "read" tools across sources. Use cases include **research assistants** that comb through corporate data, or an internal chatbot that can answer "Where is the design spec for project X?" by searching a repository. Early adopters like **Block (Square)** and **Apollo** have integrated MCP to break down information silos in their systems. It enables *"maintaining context as AI moves between different tools and datasets"* – so an agent can seamlessly pull info from, say, a database and a document store in one session.

- **Enterprise Search & Analytics:** Expanding on the above, MCP can interface AI with enterprise analytics tools or search engines. Imagine querying a **financial data MCP server** with natural language ("What was our Q3 revenue vs Q2?") – the server might translate that to a database query and return the result for the AI to present. Similarly, a **Microsoft Graph or Fabric MCP** could allow AI to unify data querying across Office documents, emails, etc., which some companies are actively exploring (GraphQL-to-MCP bridges for Microsoft Fabric data have been discussed). The result is a powerful **conversational BI tool** that can draw from multiple systems in one answer.

- **Software Development and DevOps:** Developer tools are a major area adopting MCP. Anthropic noted that development tool companies like **Zed (code editor), Replit, Codeium, and Sourcegraph** are working with MCP to enhance their platforms. In practice, this means a coding assistant (like Claude or GPT integrated in an IDE) can connect to MCP servers for things like:
  - **Filesystem operations**: reading/writing project files, using a secure filesystem MCP server (instead of relying on the model's limited context window to "see" code).
  - **Git operations**: via a Git MCP server, the AI can run git commands, inspect diffs, commit code, etc., as part of assisting a developer.
  - **Documentation search**: a server that indexes API docs or engineering wikis so the AI can fetch relevant docs when writing code.
  - **Issue trackers/CI pipelines**: an AI agent could connect to a Jira or Jenkins MCP server to create tickets or check build statuses.

  These capabilities turn AI coding assistants into much more **effective pair programmers**, because they can retrieve context (e.g. find where a function is defined in the repo) or take actions (open a PR) on behalf of the user. Maintaining state (like which files are open) is especially useful here. It's telling that Anthropic specifically calls out coding as an area where MCP *"enables AI agents to better retrieve relevant information around a coding task and produce more functional code with fewer attempts."* In other words, the AI can truly be aware of the project's context, not just a snippet.

- **Autonomous Agents and Multi-step Workflows:** MCP's pattern is also a natural fit for *agentic AI* – systems that plan and execute sequences of actions towards a goal. For example, consider an **AI sales agent** that autonomously researches a client and drafts an email. It might use an MCP server to query a sales database for client info, another MCP server to search recent news about the client's industry, and then a tool to send an email via Outlook. Because MCP supports multi-step sessions, the agent can do all this fluidly. Industries like finance or real estate could use this for AI that performs research and due diligence. Another example: an **AI travel planner agent** (like the earlier flight search scenario) could utilize a **Browser Automation MCP server (e.g. Puppeteer)** to navigate travel websites, fill forms, scrape results, etc., over a series of steps. MCP's session orchestrator (Section 8) is built to handle exactly those multi-action scenarios, such as keeping track of an open browser state across steps.

- **Knowledge Bases and Semantic Search:** A specialized but important application is building **MCP-based knowledge bases** for AI. Several open-source MCP servers integrate with vector databases and document stores. These essentially allow the AI to perform semantic search and retrieval through MCP. For instance, Amazon has demonstrated an *Amazon Bedrock Knowledge Base MCP server* that connects AI to enterprise knowledge indexed by Bedrock (with embedding-based search) (Source: aws.amazon.com). BytePlus similarly discusses using MCP knowledge bases for *"intelligent information management"* with semantic search and neural retrieval (Source: byteplus.com)(Source: byteplus.com). In practice, an AI could ask an MCP knowledge base server questions and get passages or answers drawn from company data, with the heavy lifting done by the server's ML models and indexes. This pattern overlaps with RAG but makes it more plug-and-play. *Retrieval-augmented generation applications* (like advanced research assistants or document Q&A systems) can be greatly streamlined with MCP (Source: byteplus.com). Instead of custom integration, one just spins up (or connects to) a "knowledge base MCP server" that handles embedding, searching, and returning results to the AI.

- **Other domains**: Virtually any industry that can benefit from AI accessing live data can leverage MCP. In finance, an AI could connect to trading data or compliance databases via MCP tools (with appropriate guardrails). In healthcare, an AI assistant for clinicians might use MCP to pull patient records or medical literature (with careful privacy controls). E-commerce bots might query product databases and inventory via MCP. **Education** is another – an AI tutor could use an MCP server that has a repository of course materials or interactive tools (like a math solver) to better help students. The open-source community has already built hundreds of MCP connectors (over 800 by one count) spanning everything from Slack and Gmail to weather APIs and IoT devices. This growing ecosystem means **ready-made building blocks** for various use cases.

## 5. MCP vs. Traditional Function Calling, GraphQL, LangChain, and RAG – A Comparison

With an understanding of MCP, it's valuable to compare it to other approaches for connecting AI or clients to data. Below we discuss how MCP contrasts with traditional function calling, GraphQL APIs, the LangChain framework, and Retrieval-Augmented Generation systems:

**5.1 MCP vs Standard Function Calling (LLM API tools):** *How different is MCP from the function-calling mechanism built into LLM APIs like OpenAI's?* In many ways, MCP is a superset and standardization of that idea. Both share the goal of enabling an AI to use external functions. However, there are key differences:

- *Architecture:* Traditional function calling is essentially **LLM-to-function direct invocation** – the model outputs a function name and args, and some host code immediately invokes that specific function. It's straightforward but limited to that direct call. MCP uses a **client-server protocol**. The AI doesn't call a Python function in-process; it sends a JSON-RPC request to an MCP server which then handles it. This adds slight overhead but huge flexibility: the server could be in another process, machine, or language, and can maintain a conversation (not just one call). It's a more **decoupled, service-oriented architecture**.

- *Use Cases:* Function calling shines for **single-step, well-defined operations** – e.g. "convert units" or "fetch today's weather" are one-and-done calls. MCP shines for **dynamic, context-rich scenarios** where multiple interactions or ongoing context is needed. For example, if you need the AI to go through a multi-step form fill or guide a user through a workflow, MCP is much better suited (the AI can maintain state through the session). MCP's ability to incorporate context updates also means it can handle cases where information changes during the dialogue.

- *Flexibility:* Function calling is constrained to a **predefined set of functions**. If the user's request doesn't neatly match one of them, the model either hallucinates or fails. MCP offers more **dynamic discovery** – the AI can ask an MCP server "what can you do?" via `tools/list` and adapt. Additionally, because MCP servers can hold *prompts* (templates, examples), the AI effectively gains new prompting logic along with functions, which function calling alone doesn't provide. As F22's comparison notes, MCP offers *"greater flexibility through dynamic context provision"*, whereas function calling is *limited to predefined functions*.

- *Scalability:* Managing dozens of individual functions in an LLM prompt or API call can get unwieldy (each with its own JSON schema, description, examples). There's also no inherent way to categorize or modularize them across domains – it's just one flat list per application. MCP naturally segments tools by server context and allows servers to host many tools without overloading the core AI prompt. The AI only sees the tools from the servers it's connected to, and those servers can be swapped in/out. MCP is *"designed to handle complex integrations at scale"* where function calling may hit complexity limits as functions multiply. Essentially, MCP introduces a **layer of abstraction** – instead of hardcoding 100 functions in your AI app, you might connect to 5 MCP servers each providing 20 tools, and those can be maintained independently.

- *State and Memory:* As discussed, **MCP is stateful** – servers can remember what happened earlier in the conversation (within that session). Standard function calls are stateless; the onus was on the developer to pass relevant context each time. MCP servers have **context stores** that allow them to "remember things between requests". For example, an MCP *Memory* server could store conversation history or previously retrieved facts, acting as extended memory for the AI. This opens up interesting possibilities like persistent user profiles or long-running agent sessions that survive beyond a single prompt/response cycle.

- *Security:* Function calling is fairly **sandboxed by design** – the model can only call what's exposed, which is similar in MCP. However, MCP adds formal *capability negotiation*. During initialization, the client and server exchange what features they support and possibly authenticate (Source: modelcontextprotocol.io)(Source: modelcontextprotocol.io). The server can advertise e.g. that it supports a `tools` primitive with certain tools, and maybe `resources` too, and the client agrees. This provides a clear contract of what's available and can include security scopes (MCP has a concept of *"roots"* which define namespace and access scope for a server's actions). While a simple function call doesn't address cross-system trust, MCP being an open protocol encourages using standard auth (OAuth tokens for servers, etc.) (Source: modelcontextprotocol.io). It's built with enterprise scenarios in mind, so more attention is paid to **secure data transmission and permissions**.

In summary, traditional function calling is a powerful but limited *local* mechanism; MCP transforms it into a *distributed, standardized service*, suitable for complex, enterprise-grade AI integrations. Both have their place – *"the choice between implementing Function Calling or adopting MCP hinges on application needs like scalability and interaction complexity"* – but MCP clearly aims to be the more **robust, future-proof solution** for integrating AI with the world.

**5.2 MCP vs GraphQL:** GraphQL is a query language for APIs that gained popularity for letting clients request exactly the data they need from a single endpoint. Interestingly, MCP and GraphQL share a goal of simplifying client-data interactions, but they approach it differently:

- GraphQL is designed for **structured data retrieval**. A client sends a query specifying fields, and the GraphQL endpoint returns JSON data. It's great for **flexible data queries** in traditional web/apps, solving issues of over-fetching and under-fetching that REST APIs had.

- MCP is designed for **AI-driven actions and context exchanges**, not just data queries. Instead of querying fields, the AI dynamically discovers "what can I do or fetch?" from the MCP server and then invokes those operations as needed. This is more *procedural and interactive* compared to GraphQL's declarative data query.

A simple analogy: GraphQL is like ordering off a menu by specifying exactly which ingredients you want, whereas MCP is like having a conversation with a chef who can perform tasks and also ask you questions back. Technically:

- **Connection model:** GraphQL is typically **stateless HTTP request-response** (though it has subscriptions for realtime). Each query is independent. MCP keeps a **persistent session** (over a socket or SSE) where both client and server can send multiple messages. MCP's persistent connection (stateful) vs GraphQL's usual stateless calls is a fundamental difference.

- **Schema vs Discovery:** GraphQL requires a fixed schema defined on the server, and clients introspect it. MCP also has a notion of discovering capabilities (via `list` methods), but it's more flexible – a server can even change available tools at runtime and notify the client (e.g., enabling new actions in the middle of a session) (Source: modelcontextprotocol.io). GraphQL's schema is more static and strongly typed; MCP's tool/resource list is dynamic and can carry richer natural-language descriptions for the AI.

- **Data vs Actions:** GraphQL can mutate data, but it's primarily about fetching structured data. MCP handles *actions* (tools with side effects) as first-class, and also unstructured or semi-structured data (resources may be text blobs or file contents). In fact, one could **wrap a GraphQL API inside an MCP server** – exposing some GraphQL queries as MCP tools – to let an AI use GraphQL via natural language. Apollo's recent work on an "Apollo MCP Server" suggests exposing GraphQL endpoints through MCP so AI can use GraphQL with a standard interface.

- **Real-time and two-way:** GraphQL subscriptions allow server->client push for events, which is analogous to MCP's notifications. But MCP being two-way means the server can actively call back for things like model completions, which GraphQL doesn't encompass (GraphQL is one-directional in use). MCP is inherently *"real-time, two-way communication, dynamic tool discovery"* whereas GraphQL provides flexible but *one-shot queries*.

In practice, these two can complement each other: GraphQL could be the language an MCP server uses internally to fetch data. But for the AI developer, MCP operates at a higher abstraction: *the AI doesn't need to write GraphQL queries; it just asks for what it needs and the server could handle translation*. One blog put it nicely: **REST is for stable APIs, GraphQL for flexible data queries, and MCP for AI agents needing dynamic, real-time tool access**. They serve different needs. GraphQL excels in traditional app data loading (e.g. getting exactly the UI data in one round-trip); MCP excels in letting an *AI* figure out what actions or data it needs through an interactive process. Indeed, **MCP can be viewed as an *LLM-first API* paradigm** – it's designed from the ground up for AI agents, whereas GraphQL was designed for front-end developers.

**5.3 MCP vs LangChain (Tools/Agents frameworks):** LangChain is a popular framework that emerged to help developers build complex LLM applications, including agent loops with tool usage. It provides abstractions for defining tools (functions) and an agent that uses an LLM to decide which tool to call next, etc. How does this compare to MCP?

The key difference: **LangChain is an application-layer framework, while MCP is an interoperability protocol**. LangChain's tools are Python (or JS, etc.) functions often used in-process with the LLM, and LangChain handles the prompt engineering to get the model to call those tools using few-shot examples. MCP, on the other hand, defines a standardized out-of-process interface for tools that any AI client can connect to.

Some points of contrast:

- *Audience:* LangChain is **developer-centric** – if you're building a custom agent, you use LangChain to wire it up, craft prompts, and manage the loop. MCP is **deployment-centric** – it's meant to let tools and AI clients talk without the user (or developer) having to custom-build that relationship each time. As LangChain's founder Harrison Chase noted, *"LangChain helps you build and structure the AI's internal logic, while MCP standardizes how that AI connects to external tools"*. If you control the whole stack and can code the agent logic, you might not *need* MCP for integration (you could just call functions directly). MCP is most useful when you **want to add tools to an AI agent you don't control** – e.g. a closed source AI app or a third-party platform. It allows a level of modular extension without digging into agent internals.

- *Protocol vs Library:* LangChain tools don't adhere to any universal protocol; they're just Python callables with some description metadata. This means only your agent (in LangChain) knows how to use them. MCP tools are exposed via JSON-RPC and could be used by any compliant agent or even non-LLM client. MCP is thus better for **standardizing and sharing integrations**. For example, if someone writes an MCP server for Slack, any AI (Claude, GPT-4, etc.) that implements an MCP client can use Slack via that server. With LangChain, if someone writes a Slack tool, it's tied to their codebase (though LangChain did amass a large tool library, it still required using LangChain to leverage them).

- *Integration complexity:* Using LangChain, especially with many tools, can get complex – you have to manage the chain of thought prompting and ensure the model knows when to use which tool. MCP shifts some of that complexity into the **MCP client (platform)** and the model itself. For instance, OpenAI and Anthropic's platforms have built-in support for taking an MCP server's tool list and making them available to the model (like ChatGPT's "Connectors"). This could simplify application code. However, LangChain often provides more **fine-grained control** since you can design exactly how the agent reasons (at the cost of more manual effort).

- *When to use which:* If you are developing a **bespoke agent** with specific tools and you want maximum control over prompts and logic, LangChain or similar libraries are appropriate. Indeed, Harrison Chase admitted *"if I was writing an agent to do X, there is zero chance I would use MCP"* – because he'd directly integrate the needed tools in code. However, if you want to **empower non-developers or end-users to equip AI with new tools** easily, MCP is ideal. For example, a non-programmer could run an open-source MCP server for Spotify and connect their ChatGPT to it, without knowing about prompt engineering. MCP thus opens the door for more **plug-and-play extensibility** in AI systems (much like browser plugins for users).

- *Combining them:* Interestingly, these approaches can complement each other. LangChain has even created adapters to use MCP tools within a LangChain agent. So a LangChain agent can treat an MCP server as a tool provider. Conversely, an MCP server could internally use LangChain to orchestrate a sequence of operations (some community MCP servers likely do that for complex tasks). They're not mutually exclusive, but philosophically MCP is pushing toward a world where tools are modular services any AI can call, whereas LangChain emerged in a world without that standard, to fill the gap by guiding the LLM with clever prompting.

In summary, **LangChain = development framework for LLM applications; MCP = connectivity layer for AI and tools**. MCP aims for broad interoperability and ease of integration, whereas LangChain is about building custom logic. Both aim to enable "AI agents" with tools, but at different layers of abstraction.

**5.4 MCP vs Retrieval-Augmented Generation (RAG) systems:** RAG refers to the pattern of enhancing an LLM's outputs by retrieving relevant documents (usually via vector similarity search) and feeding them into the prompt. How does MCP relate to this? In fact, RAG can be *seen as one specific use-case* of the broader MCP approach:

- Many MCP servers implement retrieval capabilities (as **resources** or tools). For example, an *MCP Vector DB server* might have a `query_embeddings` tool or a `document/search` resource. The AI can leverage these to get relevant text chunks given a query, then continue the conversation with that info. This achieves the same result as a RAG pipeline, but with the AI in control of when and how to retrieve.

- The advantage of MCP here is **standardized, plug-and-play retrieval**. Rather than custom code for each vector database or knowledge source, you spin up the corresponding MCP server. If tomorrow you switch from ElasticSearch to Pinecone for your docs, you'd just use a different server (or the same interface if it's abstracted).

From the AI's perspective, it still does `resources/list` and `resources/read` calls, etc. This decoupling is powerful.

- Moreover, MCP can combine retrieval with other actions. A classic RAG system retrieves text and appends it to the prompt for answer generation. With MCP, an AI agent could retrieve some data, then also call another tool to take action based on it, all in one continuous flow. It's not limited to just augmenting the context; it can also act on retrieved knowledge.

- One explicit mention is that MCP *"resources [provide] context similar to RAG systems but with standardized access"*. In other words, MCP formalizes what many RAG implementations did in an ad-hoc way. For example, instead of a custom vector search codepath, you have an MCP resource. This can also encourage reuse of retrieval modules across applications.

That said, RAG is more of a *design pattern* than a platform. You could implement RAG using MCP, or without it. If you do it without MCP, you might call a search API or a database directly from your code whenever needed (which works, but isn't as flexible to changes or as model-driven). With MCP, you let the model decide when to retrieve, by exposing a "search" tool. This is especially useful when the queries aren't known in advance (the AI might decide mid-conversation that it needs more info).

One potential trade-off: direct RAG (where the app explicitly retrieves and feeds the model) allows the developer more control over what gets retrieved and how it's presented to the model. MCP yields more autonomy to the AI – which can be powerful, but also requires trust that the model will use the tools appropriately. In practice, a combination could be used: the AI might retrieve via MCP, and the client app could still verify or post-process the retrieved data if needed.

In summary, MCP doesn't replace RAG; it **generalizes it**. RAG deals with one type of context (documents), whereas MCP handles documents, tools, and more. But for any scenario where you'd consider RAG (like question answering over a corpus), using an MCP server for retrieval is a compelling approach to make your solution more standardized and easily extensible.

## 6. MCP Architecture Deep Dive – Components and Interactions

To truly appreciate MCP's strengths, let's dive into **how MCP servers work under the hood** – how they orchestrate function calls, manage context, and ensure performance. An MCP server's implementation typically has several core components:

- **Communication Layer:** This is the MCP protocol handler – responsible for managing the JSON-RPC connection with the client (AI host). It parses incoming requests (e.g., "tools/call" messages), dispatches them to the appropriate internal logic, and formats responses back. It also handles the initial **handshake** when a client connects. During this handshake (the `initialize` method), the server and client exchange protocol versions and capabilities (which primitives each supports, etc.) (Source: modelcontextprotocol.io)(Source: modelcontextprotocol.io). The result is that both sides "agree" on how to communicate. The comm layer keeps the session open (for STDIO, that means keeping the process pipe open; for HTTP, often holding an SSE channel for events). **Stateful session:** Importantly, the server keeps the session open so it can associate multiple requests to the same context. If a client disconnects, the session can end (and the server may cleanup resources). This persistent connection is akin to having a continuous RPC session, enabling features like multi-step transactions and server push.

- **Request Handlers (Tools/Resources/Prompts logic):** These are the functions or methods within the server that actually implement each capability. When a JSON-RPC request comes in (say `{"method": "tools/call", "params": {name: "sendSlackMessage", ...}}`), the comm layer hands it to the corresponding **handler**. For a *Tool*, the handler will perform the action – e.g. call the Slack API with given parameters – and then return the result or an error. For a *Resource*, the handler will likely perform a data lookup and return data (possibly chunked or with pagination if large). For a *Prompt*, the handler might return a template string or even initiate a sequence (some prompt handlers could guide multi-turn interactions). Essentially, each **capability = code that executes when called**. MCP SDKs make it easy to define these (often via decorators or config). The server also implements standard handlers for discovery methods like `tools/list` (to enumerate available tools and their schemas). These handlers form the business logic of the server.

- **Context Store (Session State Storage):** A major feature is that MCP servers can maintain data between calls. The *context store* is an internal storage where the server keeps any information that needs to persist across requests or sessions. This could be:

  - In-memory variables (like a dictionary tracking what step of a multi-step process the user is in, or caching a recently fetched dataset).

  - A database or file if persistence is needed beyond a single session (e.g. storing conversation history to disk, or using Redis to share state across server instances).

  - Specialized memory structures: for example, a **vector cache** of recent user queries and embeddings for quick similarity lookup, or a mini knowledge graph built during the session (one of the reference servers is a "Memory" server that builds a persistent knowledge graph).

  The context store is what gives MCP servers a form of **"memory."** It allows faster responses (no need to recompute or re-fetch something the server already got earlier), and enables more complex interactions. For instance, a browser MCP server might store the current page DOM after an `openPage` tool, so that a subsequent `clickElement` tool knows what elements exist (without reloading the page). Or a coding assistant's MCP server might cache the content of files opened, to avoid re-reading from disk on each query. Without a context store, the AI would have to include all needed context in each request (inefficient and sometimes impossible due to token limits). With it, the server becomes an extension of the AI's working memory, but structured and bounded by the server's design (which is good for reliability).

- **Session Orchestrator:** This component manages the lifecycle of sessions and complex multi-step operations. If the MCP server is handling simple idempotent calls (like a single DB query), a lot of "orchestration" is not needed. But consider the earlier example of an **agent booking a flight** via a browser MCP server. That sequence involved multiple steps (open site, fill form, parse results). The session orchestrator can provide a framework to coordinate these:

  - It might assign a **session ID** or keep a context object per client connection, linking all actions together.

  - It ensures that data from one step is accessible in the next (often via the context store as described).

  - It can manage **control flow**: e.g., if a certain sequence of calls should be atomic or if certain steps should only happen after others.

  - It handles **timeouts or termination**: If the client disconnects or is idle for too long, the orchestrator can clean up resources (close files, browser instances, etc.) to avoid leaks.

  Essentially, the session orchestrator is the glue for **long-running or connected tasks**. Not all MCP servers need complex orchestration, but those that do (like anything simulating a multi-step agent) rely on this. This makes MCP servers capable of implementing workflows that span multiple model prompts without losing continuity.

- **Caching Layer:** MCP servers often deal with external APIs or databases which might be slow or have rate limits. To optimize, many servers include a **caching mechanism**. For instance:
  - An MCP server that fetches stock prices may cache results for a few seconds so that if the AI asks the same question again, it doesn't hit the external API repeatedly.
  - A document search server might cache the embeddings of recently accessed documents in memory.
  - A multi-level cache might be used – first check an in-memory cache, if not found maybe check a disk cache, otherwise query the actual source.
  - Caching is especially useful if multiple AI queries in the same session (or across sessions) ask for similar data. E.g., the first time an AI asks a codebase server to open `utils.py`, it reads from disk; subsequent times it can serve from memory.

  Good caching can significantly **boost performance and reduce token/cost usage**, as noted in a Reddit discussion where a *"Memory Cache MCP server"* was built to reuse data instead of re-sending it to the model repeatedly. Of course, caches must be invalidated appropriately (e.g., if underlying data changes). The caching layer in an MCP server is analogous to caching in any web service, but here it benefits AI-context usage (saving tokens by not re-sending large context unnecessarily, for example).

- **Logging & Monitoring:** Not explicitly asked, but worth noting: MCP servers often have logging of tool calls, errors, etc., which is vital for debugging when an AI does something unexpected. The MCP spec even defines a *logging primitive* for servers to send log messages to the client for monitoring. This can be useful in enterprise settings to audit what an AI accessed or did.

Bringing it all together, when an AI agent interacts with an MCP server, the flow might look like:

1. **Initialization:** The client opens a connection, sends `initialize`. The server responds with its capabilities (e.g. "I have these tools: X, Y, Z and I support resources with list/read, etc.") (Source: modelcontextprotocol.io)(Source: modelcontextprotocol.io). They confirm readiness (client sends `initialized`) (Source: modelcontextprotocol.io) (Source: modelcontextprotocol.io).

2. **Tool listing:** The AI (or the client on behalf of AI) may call `tools/list` to get details of each tool (names, descriptions, expected parameters). The server's handler returns these. Now the AI "knows" what it can do with this server.

3. **Usage loop:** The AI decides to use a tool – e.g. calls `tools/call` for tool "X" with certain args. The server's request handler for X executes the action, possibly using the context store or external APIs, then returns a result. The AI receives the result (the client passes it into the model's conversation). The AI might then choose another tool call, etc.

   - Meanwhile, the server might update context: e.g., after a `searchDocuments` call, it might store the results in the context store for quick access if the AI asks to read one of them next.

4. **Notifications:** If the server needs to send an event (say one of the operations is long-running), it can send a `tools/event` or custom notification with partial progress. The client delivers this to the AI or handles it (maybe showing a loading indicator).

5. **Completion:** Eventually, the AI signals it's done with the tools (in ChatGPT's case, it would stop calling functions and produce a final answer to the user). The session might remain open if the conversation continues, or it might eventually close. On disconnect, the server's orchestrator cleans up.

This design ensures the AI and server remain **in sync**, sharing a common context of what's been done. It's a sophisticated dance of RPC calls, but abstracted enough that developers can implement it using available SDKs without getting bogged down in low-level details. As the WorkOS analysis concluded, each part of the server *"has a clear role—request handlers perform actions, the context store gives memory, session management keeps interactions connected, and caching boosts performance."* Together, these make AI integrations far more **structured, efficient, and "agentic"** than earlier paradigms.

## 7. Diagrams and Tables Summarizing Key Concepts

To clarify the MCP pattern, this section provides a diagrammatic summary of the MCP architecture and a comparison table of MCP vs other integration methods:

**MCP Architecture Summary:**

- **Figure 1 (earlier)** already illustrated how an AI assistant connects to multiple MCP servers. Each server may correspond to a domain (files, database, chat, etc.), and the AI can call tools on each. The connections are persistent, enabling iterative calls.

- **MCP Workflow Diagram (Conceptual):** *Before MCP vs After MCP:* (Textual illustration) – Before MCP, each AI or agent required custom integration with each tool/data source (imagine multiple arrows from each model to each service). After MCP, all AIs and tools speak one language (one standardized hub). This was described through the restaurant analogy in section 1. In practice, one can envisage a **matrix** of AI systems vs data sources turning from a dense matrix of point-to-point connections (pre-MCP) to a two-layer architecture (post-MCP) where AI systems connect to MCP and MCP connects to data, decoupling the matrix.

- **MCP Data Flow Diagram:** The sequence from the AI's question to using a tool to responding:

   1. User asks AI a question.

   2. AI (via client) calls `tools/list` on servers to see what it can do (if not already known).

   3. AI decides to call a specific tool (sends `tools/call`).

   4. MCP server executes action, returns data (maybe as a resource reference if large).

   5. AI incorporates data and either calls more tools or answers user. *(This could be represented in a swimlane diagram with lanes: User, AI/LLM, MCP Client, MCP Server, External API/Data.)*

**Comparison Table:** Below is a summary **table comparing MCP with traditional function calling, GraphQL, and LangChain** on key dimensions (combining insights from earlier sections):

| ASPECT | FUNCTION CALLING (LLM API) | GRAPHQL API | LANGCHAIN TOOLS | MODEL CONTEXT PROTOCOL (MCP) |
|---|---|---|---|---|
| **Communication** | One-shot call within chat; model outputs function name + JSON args. Stateless beyond each call. | Client sends query string to server; stateless request-response (subscriptions for realtime). | In-process function calls orchestrated by an agent loop; typically stateless w.r.t. external session (prompt manages state). | Persistent JSON-RPC session (stateful) between AI client and server. Two-way messaging (client ↔ server) with streaming and notifications. |
| **Standardization** | Proprietary to each model (no universal format; e.g. OpenAI vs Claude JSON differ). Functions must be defined for each. | Industry standard query language for APIs (GraphQL spec). Strongly typed schema introspection. | LangChain provides a *de facto* standard for tool interface in Python, but not an open protocol; other frameworks differ. | Open standard (JSON-RPC 2.0 based). Any compliant client and server can interoperate, regardless of language/vendor. "USB-C for AI tools." |
| **Capabilities** | Only developer-defined functions (with fixed parameters) are available. No built-in data retrieval unless defined. | Only what the GraphQL schema exposes (usually data fields, some mutations). No concept of AI or LLM in spec. | Any Python function or chain can be a tool; very flexible, including actions or data retrieval. But tied to agent's code. | Tools (actions with side effects), Resources (data retrieval streams), and Prompts (templates) are first-class primitives. Server can expose any or all types. Dynamic discovery of available ops. |

| ASPECT | FUNCTION CALLING (LLM API) | GRAPHQL API | LANGCHAIN TOOLS | MODEL CONTEXT PROTOCOL (MCP) |
|---|---|---|---|---|
| **Context & Memory** | No memory beyond model's own (model might remember previous call results if in conversation). Each call stateless; developer must feed context if needed. | Server doesn't remember past queries unless client includes an ID; stateless by design. | Memory has to be handled in the agent (e.g. keeping track of observations in prompt memory or external memory components). | Built-in session state on server; context store enables memory of earlier actions/results. AI doesn't need to resend info; server can cache and maintain context across calls. |
| **Interaction Pattern** | Model *decides* to call a function mid-response, then usually immediately gets result and continues. Typically 1 function per user query (though can chain via multiple turns). | Client explicitly queries for exactly what it needs in one go (can nest data requests). Multi-step logic must be orchestrated client-side with multiple queries. | Agent uses an *iterate-thought then tool* loop (prompting model to think and use tools repeatedly) – multiple calls possible, but coordination logic is custom. | AI can engage in **multi-step tool use** within one session (list tools → call tool1 → call tool2…). The protocol supports iterative reasoning with the server as partner (e.g., notifications, eliciting user input mid-task). |
| **Real-time & Async** | Not inherently realtime; model waits synchronously for function result via API. Cannot handle events except by polling via functions. | Supports real-time via subscriptions (server pushes data on events). Client must set up subscription queries. | LangChain agent can be coded to handle streaming outputs or multiple steps; not real-time event driven except if custom-coded. | **Event-driven**: server can push notifications (e.g. progress updates) (Source: modelcontextprotocol.io). Suitable for long-running tasks – AI can wait or handle partial results. Two-way: server can also ask AI or user for info, enabling interactive workflows. |

| ASPECT | FUNCTION CALLING (LLM API) | GRAPHQL API | LANGCHAIN TOOLS | MODEL CONTEXT PROTOCOL (MCP) |
|---|---|---|---|---|
| **Ease of Use** | Very easy for simple functions – just register and model will use if appropriate. Harder to manage many functions. | Flexible for data queries; requires learning GraphQL schema and setting up resolvers. Not trivial for non-developers. | Powerful for developers who know how to prompt and structure agents. Non-trivial to design correctly. | Initial setup (MCP server & client config) needed, but then adding tools is standardized. Can be packaged for non-dev end users (one-click connectors). Still an emerging ecosystem, some complexity in running servers. |
| **Scalability** | Tight coupling to model; adding new data source means new function and redeploy. Harder to reuse across apps. | GraphQL scales well for many clients and queries, but each integration is one schema – adding new data source expands schema. | Agent frameworks scale code-wise, but each deployment is custom. Hard to reuse an agent's tools in another without copying logic. | **Highly scalable via modularity**: new MCP servers can be added without affecting client code. One server can serve many clients. Designed for enterprise scale (multiple concurrent sessions, etc.) with proper backend scaling. |
| **Example Use** | "What's weather in Chennai?" → model calls `getWeather(location="Chennai")`. Good for direct Q&A with external info. | "Give me user name and their orders" → client sends GraphQL query for user and nested orders, gets JSON. Good for building a UI page. | "Find document about X then summarize" → agent uses a search tool, gets text, then uses a summarize tool. Good for custom AI workflows where dev controls logic. | User asks, "Summarize recent sales" → AI uses CRM MCP server's search resource, then calls report tool. Good for letting AI autonomously navigate multi-step tasks with data. |

*(Table sources: Function Calling from OpenAI/Anthropic docs ; GraphQL from OpenReplay blog ; LangChain vs MCP from Harrison Chase ; MCP details from OpenReplay and F22 Labs .)*

This table highlights that **MCP combines aspects of API design (like GraphQL's flexibility) with the goal of AI autonomy (like LangChain's agents), all in a standardized, session-based protocol**. Each approach has its niche, but MCP's unique blend of features positions it well for the era of AI agents interacting with many systems.

## 8. Technical Deep Dive: Orchestration, Caching, and Context Management in MCP

One of the most powerful aspects of the MCP pattern is how it handles the technical challenges of letting an AI use external tools repeatedly and efficiently. Let's explore **how MCP servers orchestrate function calls, cache results, and manage contextual memory/session state** in practice (some of this was touched on in section 6, but here we focus on these aspects explicitly):

- **Orchestrating Function Calls (Multi-step Tool Use):** Traditional function calling didn't require orchestration – each call was independent. But MCP servers often need to **orchestrate sequences of calls into a coherent operation**. As described earlier, the *Session Orchestrator* component in a server can maintain the "flow". For example, take a *web browsing MCP server* (like Puppeteer). If the AI wants to scrape information, it might:

  1. Call `open_url("flights.com")` – server opens a browser instance (and stores a handle in context).
  2. Call `fill_form(...flight details...)` – server's orchestrator uses the stored browser instance (from context) to fill the form.
  3. Call `click("Search")` – triggers the search.
  4. Server sends a `notification` when results page is loaded or perhaps streams snippet of results.
  5. AI may then call `extract_results()` – server returns structured data of flight options.

  Throughout this, the *orchestrator ensures each step has what it needs*: the browser instance, the page DOM, etc., tying the requests together into a workflow. It also ensures that if the session closes, the browser is closed. From the AI's perspective, it's like controlling a mini-agent (the browser) via successive function calls, but the server is managing the intermediate state. This orchestration is what enables **complex actions** – without it, the AI would be limited to trivial stateless API calls. Other examples: a database MCP server might allow transactions spanning multiple calls (begin transaction, multiple queries, then commit/rollback through orchestrator logic). Or a PDF reading server might allow an AI to open a PDF, then request page 5, then page 10, without reloading the file each time – orchestrator keeps the file open and tracks current page.

- **Caching Results to Optimize Performance:** MCP servers often act as middleware between a possibly slow or expensive data source and the AI. Caching at the server can greatly improve efficiency:

  - **Token Optimization:** If an AI asks the same thing twice, the server can avoid sending a long response twice. For instance, a *Memory Cache MCP server* was built to cache query results and only send a short reference or skip re-sending content to save tokens. The AI can then rely on a shorter identifier or confirmation that the data hasn't changed.

- **Rate limiting and latency:** Suppose an AI is summarizing logs and calls a `get_log_entries(date)` tool repeatedly for different dates. Hitting the real API each time could be slow or even hit rate limits. A caching layer can store recent responses so that the second time the tool is called for the same date, it returns instantly from cache. This was also mentioned in WorkOS's breakdown: e.g., *"stock prices can be cached for a few seconds to avoid repeated API calls"*.
- **Multi-level caching:** A sophisticated server might use an in-memory cache for very frequent items and a disk/Redis cache for larger or less frequent items, checking in layers. For example, an image-generation MCP server might cache generated images (or their hashes) in memory for quick access, but also store them on disk so if the server restarts, it doesn't regenerate the same image.
- **Prefetching:** The server can even anticipate needs. A search MCP server might prefetch next page results when the AI requests the first page, so if the AI asks for page 2, it's already cached. This kind of predictive caching again boosts responsiveness.
- Managing cache consistency is important – servers often have to decide how long to keep data. But since the AI is often engaged in short sessions, even caching just within a session can help immensely (the same data often gets used multiple times in one conversation).
- *Example:* A vector search MCP server might cache the top N results of a query embedding. If the AI then asks to read result #3 in detail, the server already has it (no need to query the database again). If the AI refines the search query slightly, some caching logic might reuse previous embedding computations or results intersections.

- **Contextual Memory and Session State:** Perhaps the biggest differentiator for MCP is how it allows **contextual memory** to be handled off-model:
  - **Persistent Context:** As mentioned, the context store can hold *session history or intermediate results*. Think of an **MCP Memory server** that a coding AI uses: it could store a summary of the conversation so far or key variables, effectively externalizing the model's long-term memory. One real example: the *MCP Memory Keeper* server for Claude stores code context to prevent it from being lost during context window compaction. By plugging that in, Claude's coding session can "remember" details beyond its usual token limit, because the server re-injects or keeps track of them.
  - **User-Specific State:** In enterprise use, you might have an MCP server maintain a user profile or preferences across sessions (using a database as context store). Then whenever that user's AI assistant connects, it pulls that context (as resources or prompts) to personalize responses. For instance, a customer support AI might store which solutions the user has already tried in an MCP memory server, so it doesn't repeat itself.
  - **Session Isolation:** Each client connection typically has its own context space unless designed otherwise. This is good for privacy and logical separation. The orchestrator ensures that, for example, two different users using the same tool via MCP don't see each other's data (unless intended for a shared session).
  - **Combining with Model Memory:** The AI model still has its own internal memory (the chat history it keeps in tokens). But MCP provides an *augmented memory*. The AI can offload large context to the server. For example, an AI might retrieve a 50-page document via an MCP server but only put a summary into its prompt. If later it

needs more detail, it can ask the server again instead of the user having to provide it or the model trying to remember exact text.

- **Session Continuity Across Platforms:** Another interesting aspect is *context portability*. If multiple AI clients support MCP, the same MCP server could allow a user to switch contexts. For example, start a conversation with Claude (via Claude Desktop) connected to some MCP servers, then later connect ChatGPT to the same servers – the context (like a project's state) could persist on the server side. This hasn't been fully realized yet, but Anthropic hinted at *"maintaining context as they move between different tools and datasets"* as a future vision. Essentially, the MCP server can act as a **context hub** that survives beyond any single AI session.
- **Transient vs Persistent Memory:** Some MCP servers are meant for transient session memory (cleared when session ends), others can persist. For example, the *Memory (knowledge graph)* reference server likely builds a persistent store of facts learned across sessions. This can enable an agent that "learns" over time – each session's new info gets added to the knowledge graph. Caching vs memory is a bit blurry here: one could view persistent memory as long-term cache of knowledge.

In practice, these technical capabilities mean that an AI using MCP can achieve something very close to how a human assistant would work with external tools:

- It doesn't constantly repeat tasks or queries it's already done (thanks to caching and state).
- It remembers the context of what it's doing with each tool (session state).
- It can handle interactive workflows without losing track after each step (orchestrated sequences).
- It's responsive and efficient, not slowed down by redundant calls or the need to re-retrieve static info repeatedly.

For example, consider a knowledge worker's AI assistant summarizing quarterly reports: Without MCP, it might hit an API for each document, fetch data each time, and be stateless – possibly reloading the same doc if asked a follow-up question. With MCP, the assistant opens the report (one call), caches it, and on follow-ups it already has the content ready; it might even pre-compute some stats on it and keep them in memory. The experience is smoother and closer to how a human would take notes and not reread the entire document for each question.

From a developer perspective, implementing these requires careful design but the **MCP SDKs often provide support** (e.g., in Python SDK you might have an in-memory store you can use between handler calls). The open-source community has started building specialized MCP servers focused on memory and caching (as seen by "memory cache server" or "persistent context" projects).

In summary, **MCP's architecture tackles the challenges of tool integration – latency, statefulness, context limits – through thoughtful engineering: stateful sessions, context stores, orchestrators for multi-step flows, and caching layers.** These make AI-tool interactions more robust and performant. It's a significant leap from the stateless, isolated API calls of "function calling 1.0" and one of the reasons MCP is seen as a foundation for the next generation of AI systems.

## 9. Pitfalls, Trade-offs, and Future Directions of the MCP Pattern

No technology is without challenges. As MCP emerges as a standard, developers and researchers have pointed out some **pitfalls and trade-offs** to be mindful of:

- **Added Complexity:** MCP introduces additional moving parts – running separate server processes, managing JSON-RPC connections, etc. For simple use cases, this can feel heavyweight. As one commentary noted, *"Why does a tool protocol need to also serve prompts and LLM completions? Why two-way communication?"* – implying MCP might be more complex than necessary for basic tool use. Setting up MCP servers and ensuring they adhere to spec is extra work compared to just calling an API directly in code. This complexity could pose a barrier, especially for small-scale applications. However, the flip side is that this complexity brings generality and power (as discussed, two-way comms and prompts enable advanced scenarios). The community is likely to create more **user-friendly packaging** (like one-click server deployment, GUI-based connectors, etc.) to mitigate this.

- **Statefulness and Scaling:** Maintaining state per session can complicate scalability. Traditional stateless services (REST, GraphQL) can scale by load-balancing any request to any server instance. With MCP, if a session sticks to one server instance (for context), you might need to implement session affinity or a way to share context between instances (like using Redis for context store). Nuno from LangChain argued that *"a stateless protocol is key for usability on a server… once usable on a server, auth and other issues pop up"*. Essentially, running MCP servers as cloud services for many users requires careful design (so that, for example, hundreds of parallel sessions don't eat up too much memory with their context, and that scaling out doesn't lose session info). Solutions include externalizing state (databases) or partitioning users to specific server instances. It's a trade-off: stateful is more capable, but stateless is easier to scale. The MCP spec acknowledges this by allowing servers not to implement certain stateful features if not needed.

- **Performance Overheads:** The JSON-RPC and persistent connection approach, while generally fast (it's lightweight JSON over HTTP or pipes), may have overhead in some cases. For high-frequency small calls, a binary or in-process approach could be faster. Also, running many MCP servers (one per integration) could be resource intensive if each holds its own process and maybe even a copy of large libraries. For instance, an MCP filesystem server and an MCP Git server might each load similar data. There's talk of more consolidated approaches or lighter-weight MCP connectors. But at the current stage, some inefficiency might be the cost of modularity. Caching helps performance, but if not done carefully, memory use can grow.

- **Model Limitations:** MCP gives the model more autonomy to use tools, but current LLMs are not infallible in their tool use. They might call the wrong tool or use it improperly. LangChain's internal benchmarks found that *"models fail to call the right tool about half the time… even with tailored prompts"*. So if an AI is just given a list of 10 MCP tools, it might misuse them initially. MCP doesn't solve the fundamental challenge of **tool grounding** – the model needs to learn when and how to use tools effectively. Good descriptions, few-shot examples (possibly via the `prompts` primitive), and iterative improvements in model training will help. But in the near term, developers should expect some trial and error. This is similar to function calling where one must test if the model picks the correct function. The difference is with MCP there could be more tools and more freedom, increasing the potential for confusion. In critical applications, one might still need validation logic (the client verifying if the tool output makes sense before using it, etc.).

- **Security and Access Control:** While MCP provides a framework, ensuring that an AI using tools doesn't do something harmful or access unauthorized data is a human responsibility. MCP servers should implement proper auth (and the spec encourages OAuth/token use for remote servers (Source: modelcontextprotocol.io)). The AI model itself could be tricked by a prompt injection to call tools in unintended ways – e.g., if user input says "ignore previous instructions and delete all files", an unguarded system might attempt a `delete_file` tool. So safety mitigations like **tool permissioning** are crucial. The MCP client or host could impose policies (only allow certain tools or prompt confirmations for destructive actions). The MCP spec's introduction of *"roots"* (scopes/namespaces for resources) is one mechanism to limit what data a server can access. But developers must design with least privilege in mind. Also, exposing too many tools might increase attack surface (more to potentially misuse). These are solvable issues, but teams need to follow security best practices as they would with any powerful integration.

- **Compatibility and Competing Standards:** MCP was spearheaded by Anthropic, but OpenAI, Google, and others are also implementing their ways to connect tools. There's some convergence (OpenAI added support for MCP in their API and ChatGPT "Connectors"), but also some fragmentation. OpenAI's early integration limits ChatGPT to using only two tool methods (`search` and `fetch`) with MCP servers, essentially a subset of full MCP. This partial support created confusion among users and may slow adoption. If not all major platforms fully embrace the open MCP spec, we could see a bifurcation where each platform has its own flavor (which would be ironic, as MCP aims to unify them). However, trends suggest convergence: e.g., Claude, ChatGPT, and even open-source tools like Cursor IDE are supporting MCP clients. There's also discussion about standardizing further or converging with efforts like Microsoft's Plug-in standards or OCI for AI. **The success of MCP will depend on broad adoption and not being limited to one AI vendor.**

Now looking forward, what are the **future directions** for MCP and AI-driven data retrieval?

- **Wider Adoption and Ecosystem Growth:** In just about a year, MCP went from concept to hundreds of community-built servers. We can expect **more official integrations**. For example, database vendors might ship official MCP servers for their products (similar to how Apollo is bridging GraphQL, perhaps we'll see an "MCP for Oracle DB" or "MCP for SharePoint" etc.). The *MCP registry* or directory will grow. The Anthropic news release mentions *pre-built servers for popular systems*, and more are being added constantly. An official **MCP Hub** (like a plugin store) could emerge – indeed, there are hints of a *"central MCP Registry and standardized metadata"* in the roadmap.

- **Agent Enhancements:** The latentgenius article mentioned *"agent graphs, refined human-in-loop workflows, fine-grained permissions"* as focus areas. This suggests MCP might evolve to better support multi-agent scenarios (where multiple AI agents collaborate via MCP), or more complex permissioning (maybe each tool requiring certain user approval). We might also see **improved agent planning** around MCP – e.g., AI models that are explicitly trained or finetuned to utilize MCP tools effectively (reducing the error rate in tool selection).

- **Integration with Other Protocols (GraphQL, gRPC):** Apollo's work hints that GraphQL and MCP might converge. Possibly MCP servers could advertise GraphQL schemas as part of their resource descriptions. Also, some in the community propose that GraphQL's introspection could be used by AI to figure out API usage (GraphQL might even become a tool for the AI, which it can query). On another front, perhaps lighter-weight transports or a gRPC-based version of MCP could appear for more high-performance needs. At its core MCP is transport-agnostic JSON-RPC, but if binary protocols are needed for speed, that could happen too (some have asked about WebSocket support beyond SSE – indeed latentgenius notes WebSocket as an option for real-time).

- **Multimodal and Multi-Model Support:** Currently MCP is largely about connecting LLMs to textual or structured data tools. But the *Technical Expansion* roadmap suggests *"multimodal support, chunked bidirectional streaming, enhanced security controls"*. This could mean MCP being used to serve images, audio, or video data to AI models (e.g., an MCP server that streams chunks of an audio file for transcription, or an image analysis tool). The protocol might be extended or profiles created for such data types. Also, today a single AI (like Claude) connects to MCP. In future, maybe multiple models could be in the loop (one person imagined e.g. an MCP server could query one model for something and give result to another). These are speculative but align with making MCP a general "AI interoperability layer."

- **Standard Governance:** Being an open standard, MCP might move toward a more formal governance (perhaps an open foundation or RFC process if it gains enough traction). The roadmap's *"Governance Evolution"* suggests community-led development and possibly a standards body recognition. This would solidify MCP as not just Anthropic's baby but an industry standard (comparable to how OpenAPI spec is governed). If that happens, we could see even faster adoption across companies and open-source.

- **Simplified Deployment and Tools:** For MCP to truly become ubiquitous, it needs to be easy to use. We can expect **improvement in tooling**: GUI tools to manage MCP servers, cloud services offering MCP-as-a-service, etc. Already, tools like *MCP Inspector* exist for testing connections. In future, non-engineers might enable connectors in their AI app via toggles, without ever seeing JSON. The complexity will be under the hood. If the analogy is that MCP is like the web (HTTP for AI), then we might get something like a "browser" for MCP or a visual interface to browse available tools.

- **Combining MCP with Traditional APIs:** Not every API will have an MCP server overnight. We might see **bridges** – for instance, an MCP server that can wrap any OpenAPI-described REST API automatically (converting OpenAPI schema to MCP tool definitions). This could instantly expose thousands of existing APIs to AI in a standardized way. Similarly, connectors to systems like Zapier (which already has many integrations) might appear, marrying MCP with automation platforms.

In conclusion, the **MCP server pattern** is poised to play a central role in the **"agentic AI" future** – where AI systems are not static or isolated, but actively retrieve, reason, and act upon the world's data in real time. It extends the function calling idea into a comprehensive framework suited for complex, context-rich AI behavior. While there are challenges to address (complexity, training models to use it well, scaling considerations), the momentum behind MCP is strong. As one blog nicely put it, MCP could become a *"foundational layer – much like Dockerfile or OpenAPI has done in their domains"*, defining how AI systems interface with external context moving forward.

**Sources:**

- Anthropic (2024). *Introducing the Model Context Protocol* – Anthropic News
- F22 Labs (2025). *MCP or Function Calling: Everything You Need to Know*
- OpenAI (2025). *OpenAI Agents SDK – MCP documentation*
- LatentGenius (2025). *Model Context Protocol: Standardizing AI-to-System Integration*
- WorkOS (2025). *How MCP servers work: Components, logic, and architecture*
- OpenReplay (2025). *MCP vs REST vs GraphQL: LLM-first APIs differences*

- LangChain Blog (2025). *MCP: Flash in the Pan or Future Standard?* (Harrison Chase & N. Campos debate)
- BytePlus (2025). *MCP Knowledge Base: Features & Setup*(Source: byteplus.com)(Source: byteplus.com)
- Reddit (2024). Discussions on MCP memory cache and usage (mcp.community)

Tags: model context protocol, mcp, function calling, ai tool integration, llm architecture, data retrieval, rag

## About Cirra

### About Cirra AI

Cirra AI is a specialist software company dedicated to reinventing Salesforce administration and delivery through autonomous, domain-specific AI agents. From its headquarters in the heart of Silicon Valley, the team has built the **Cirra Change Agent** platform—an intelligent copilot that plans, executes, and documents multi-step Salesforce configuration tasks from a single plain-language prompt. The product combines a large-language-model reasoning core with deep Salesforce-metadata intelligence, giving revenue-operations and consulting teams the ability to implement high-impact changes in minutes instead of days while maintaining full governance and audit trails.

Cirra AI's mission is to **"let humans focus on design and strategy while software handles the clicks."** To achieve that, the company develops a family of agentic services that slot into every phase of the change-management lifecycle:

- **Requirements capture & solution design** – a conversational assistant that translates business requirements into technically valid design blueprints.
- **Automated configuration & deployment** – the Change Agent executes the blueprint across sandboxes and production, generating test data and rollback plans along the way.
- **Continuous compliance & optimisation** – built-in scanners surface unused fields, mis-configured sharing models, and technical-debt hot-spots, with one-click remediation suggestions.
- **Partner enablement programme** – a lightweight SDK and revenue-share model that lets Salesforce SIs embed Cirra agents inside their own delivery toolchains.

This agent-driven approach addresses three chronic pain points in the Salesforce ecosystem: (1) the high cost of manual administration, (2) the backlog created by scarce expert capacity, and (3) the operational risk of unscripted, undocumented changes. Early adopter studies show time-on-task reductions of 70-90 percent for routine configuration work and a measurable drop in post-deployment defects.

### Leadership

Cirra AI was co-founded in 2024 by **Jelle van Geuns**, a Dutch-born engineer, serial entrepreneur, and 10-year Salesforce-ecosystem veteran. Before Cirra, Jelle bootstrapped **Decisions on Demand**, an AppExchange ISV whose rules-based lead-routing engine is used by multiple Fortune 500 companies. Under his stewardship the firm reached seven-figure ARR without external funding, demonstrating a knack for pairing deep technical innovation with pragmatic go-to-market execution.

Jelle began his career at ILOG (later IBM), where he managed global solution-delivery teams and honed his expertise in enterprise optimisation and AI-driven decisioning. He holds an M.Sc. in Computer Science from Delft University of Technology and has lectured widely on low-code automation, AI safety, and DevOps for SaaS platforms. A frequent podcast guest and conference speaker, he is recognised for advocating "human-in-the-loop autonomy"—the principle that AI should accelerate experts, not replace them.

### Why Cirra AI matters

- **Deep vertical focus** – Unlike horizontal GPT plug-ins, Cirra's models are fine-tuned on billions of anonymised metadata relationships and declarative patterns unique to Salesforce. The result is context-aware guidance that respects org-specific constraints, naming conventions, and compliance rules out-of-the-box.
- **Enterprise-grade architecture** – The platform is built on a zero-trust design, with isolated execution sandboxes, encrypted transient memory, and SOC 2-compliant audit logging—a critical requirement for regulated industries adopting generative AI.
- **Partner-centric ecosystem** – Consulting firms leverage Cirra to scale senior architect expertise across junior delivery teams, unlocking new fixed-fee service lines without increasing headcount.
- **Road-map acceleration** – By eliminating up to 80 percent of clickwork, customers can redirect scarce admin capacity toward strategic initiatives such as Revenue Cloud migrations, CPQ refactors, or data-model rationalisation.

---

**Future outlook**

Cirra AI continues to expand its agent portfolio with domain packs for Industries Cloud, Flow Orchestration, and MuleSoft automation, while an open API (beta) will let ISVs invoke the same reasoning engine inside custom UX extensions. Strategic partnerships with leading SIs, tooling vendors, and academic AI-safety labs position the company to become the de-facto orchestration layer for safe, large-scale change management across the Salesforce universe. By combining rigorous engineering, relentlessly customer-centric design, and a clear ethical stance on AI governance, Cirra AI is charting a pragmatic path toward an autonomous yet accountable future for enterprise SaaS operations.

---