

# Model Context Protocol: Architecture, Design Principles, and Implications for Interoperable AI Systems

Furkan Tercan

**Abstract**—The Model Context Protocol (MCP) is an open standard introduced by Anthropic in November 2024 that defines a structured communication layer between large language model (LLM) hosts and external tools, data sources, and services. By decoupling AI models from their integrations through a unified protocol, MCP aims to address the fragmentation that arises when each application implements its own ad-hoc integration layer. This paper presents a comprehensive analysis of MCP’s architecture, its core primitives (tools, resources, and prompts), transport mechanisms based on JSON-RPC 2.0, and security considerations. We compare MCP against existing AI integration paradigms, examine early ecosystem adoption, and discuss the protocol’s implications for building context-aware, interoperable, and extensible AI systems. Our analysis suggests that MCP represents a meaningful step toward standardized AI-tool interaction, though open challenges remain in areas of authorization, versioning, and multi-agent orchestration.

**Index Terms**—Model Context Protocol, MCP, large language models, AI integration, JSON-RPC, tool use, context management, interoperability, AI agents.

## 1 INTRODUCTION

THE rapid proliferation of large language model (LLM) applications has exposed a critical infrastructure problem: each AI-powered product builds its own bespoke integration layer to connect models with external data sources, APIs, and tools. This fragmentation creates duplicated engineering effort, inconsistent security boundaries, and integrations that cannot be reused across models or hosts.

In November 2024, Anthropic introduced the *Model Context Protocol* (MCP) [1], an open standard designed to provide a universal interface between AI models and the broader software ecosystem. MCP draws inspiration from the Language Server Protocol (LSP) [2], which similarly transformed IDE tooling by standardizing communication between editors and language analysis servers. Where LSP solved the  $M \times N$  integration problem for developer tools, MCP targets the analogous problem for AI agents and their data sources.

This paper makes the following contributions:

- A formal analysis of MCP’s layered architecture and communication model.
- An examination of MCP’s three core primitives—tools, resources, and prompts—and their role in structuring model context.
- A comparison of MCP with prior and concurrent AI integration approaches.
- A discussion of open challenges in security, authorization, and multi-agent coordination.

The remainder of this paper is organized as follows. Section 2 provides background on LLM integration approaches. Section 3 describes MCP’s architecture in detail. Section 4

covers core protocol primitives. Section 5 discusses transport mechanisms. Section 6 addresses security considerations. Section 7 compares MCP with related work. Section 8 summarizes ecosystem adoption. Section 9 identifies open challenges, and Section 10 concludes.

## 2 BACKGROUND

### 2.1 LLM Tool Use and Function Calling

Modern LLMs such as GPT-4 [3], Claude [4], and Gemini [5] support *function calling* or *tool use*, mechanisms that allow a model to request the execution of external functions during inference. These capabilities enable models to retrieve live data, perform calculations, or interact with services beyond their training knowledge cutoff.

However, function-calling interfaces are model-specific and application-specific. A tool written for OpenAI’s function-calling schema requires adaptation to work with Claude’s tool use API, and neither schema is reusable across applications without reimplementation. This creates a tightly coupled integration landscape where tools, models, and applications are tangled together.

### 2.2 The $M \times N$ Integration Problem

Consider  $M$  AI applications each needing to integrate with  $N$  external services. Without a shared protocol, each integration is independently implemented, resulting in  $M \times N$  custom connectors. MCP addresses this by inserting a protocol layer: applications implement one MCP client, services implement one MCP server, and the total integration surface reduces to  $M + N$ .

---

• F. Tercan is an independent researcher.  
E-mail: furkan@example.com  
Manuscript received February 23, 2026.

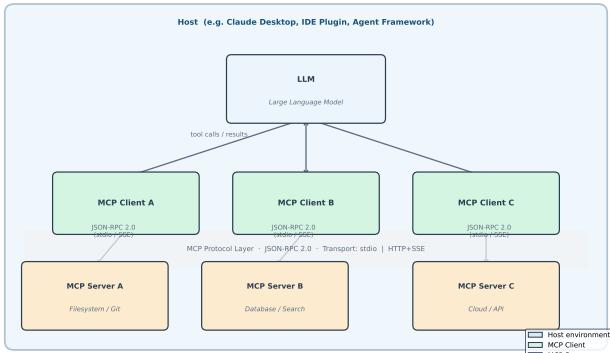


Fig. 1. MCP deployment architecture showing the host, multiple clients, and their connected servers.

### 2.3 The Language Server Protocol Analogy

The Language Server Protocol (LSP), introduced by Microsoft in 2016, solved an identical structural problem in software development tooling. Before LSP, every IDE had to implement language-specific features (autocomplete, goto-definition, diagnostics) for every programming language independently. LSP standardized the editor-to-language-server interface, enabling any LSP-compliant editor to work with any LSP-compliant language server. MCP's designers explicitly acknowledge this analogy [1], positioning MCP as "LSP for AI."

## 3 MCP ARCHITECTURE

### 3.1 Roles and Components

MCP defines three primary roles in a deployment:

- **Host:** The LLM application or agent environment (e.g., Claude Desktop, an IDE plugin, or an autonomous agent framework). The host manages user interaction, holds the LLM context, and coordinates one or more MCP clients.
- **Client:** A protocol client instantiated by the host, maintaining a 1:1 connection with a single MCP server. Each client handles message routing, capability negotiation, and session lifecycle for its server.
- **Server:** A lightweight process or service that exposes capabilities (tools, resources, prompts) via the MCP protocol. Servers are typically scoped to a specific domain—a filesystem server, a database server, a web search server—and can be implemented in any language.

Figure 1 illustrates the relationship between these components.

### 3.2 Protocol Layers

MCP's design separates concerns into distinct layers:

- 1) **Transport Layer:** Handles the physical transmission of messages. MCP supports stdio (for local process communication) and HTTP with Server-Sent Events (SSE) (for remote communication).

- 2) **Protocol Layer:** Implements the JSON-RPC 2.0 framing, including request/response correlation, notifications, and error handling.
- 3) **Application Layer:** Defines the MCP-specific message types for capability discovery, tool invocation, resource retrieval, and prompt management.

### 3.3 Session Lifecycle

An MCP session begins with an *initialization handshake* in which client and server exchange their protocol versions and supported capabilities. This negotiation allows clients and servers at different protocol versions to interoperate gracefully. Following initialization, the session enters an operational phase where the client may send requests and receive responses or notifications. Sessions are terminated either by explicit disconnect or connection failure.

## 4 CORE PRIMITIVES

MCP defines three categories of primitives that servers can expose to clients.

### 4.1 Tools

Tools are executable functions that the LLM can invoke to perform actions or retrieve computed results. Each tool is described by a name, a natural-language description, and a JSON Schema defining its input parameters. During inference, the model may decide to call a tool; the host executes the call through the MCP client and returns the result as part of the model's context.

A tool definition takes the following form:

```

1  {
2    "name": "search_web",
3    "description": "Search the web for current
4      information.",
5    "inputSchema": {
6      "type": "object",
7      "properties": {
8        "query": {
9          "type": "string",
10         "description": "The search query."
11       }
12     },
13     "required": ["query"]
14   }
}

```

Listing 1. Example MCP tool definition in JSON.

Tools are *model-controlled*: the LLM determines when and how to invoke them. This distinguishes tools from resources, which are application-controlled.

### 4.2 Resources

Resources represent data or content that the host application exposes to the model as context. Unlike tools, resources are *application-controlled*: the host decides which resources to include in the context window. Resources are identified by URIs and can represent files, database records, API responses, or any structured data.

Resources may be *static* (retrieved once) or *dynamic* (updated via server-sent notifications). The latter enables live context management, where the model's view of external state remains current without repeated polling.

### 4.3 Prompts

Prompts are reusable, parameterized message templates that servers can expose to hosts. They allow server operators to define best-practice interaction patterns for their domain. A filesystem server, for example, might expose a “summarize directory” prompt that instructs the model how to present directory contents to the user. Prompts are *user-controlled*: they are surfaced to the end user as commands or workflows.

### 4.4 Sampling

MCP also introduces a *sampling* primitive that allows servers to request LLM completions through the host. This enables servers to use the model’s intelligence as part of their own processing, facilitating agentic patterns where the server itself orchestrates model inference.

## 5 TRANSPORT MECHANISMS

### 5.1 Standard I/O (stdio)

For locally deployed servers, MCP uses standard input/output streams. The host spawns the server as a subprocess and communicates via its stdin/stdout channels. This transport is simple, requires no network configuration, and inherits the process-level isolation of the operating system. It is the recommended transport for desktop applications and developer tools.

### 5.2 HTTP with Server-Sent Events (SSE)

For remote or cloud-deployed servers, MCP uses HTTP as the base transport. Client-to-server messages are sent as HTTP POST requests, while server-to-client messages (responses and notifications) are delivered over a persistent SSE connection. This design accommodates server-initiated messages while remaining compatible with standard HTTP infrastructure including proxies and load balancers.

### 5.3 JSON-RPC 2.0 Framing

Regardless of transport, all MCP messages are framed as JSON-RPC 2.0 [6] payloads. Each request carries a unique identifier enabling response correlation. Notifications (messages with no expected response) are used for events such as resource updates or progress reporting.

## 6 SECURITY CONSIDERATIONS

### 6.1 Trust Boundaries

MCP’s architecture establishes three distinct trust boundaries: between the user and the host, between the host and MCP clients/servers, and between MCP servers and external services. Each boundary requires appropriate authentication and authorization controls. The MCP specification emphasizes that hosts must treat server-provided content—including tool descriptions—as potentially untrusted input.

### 6.2 Prompt Injection

A significant attack surface in MCP deployments is *prompt injection*: malicious content returned by a tool or resource that attempts to override the model’s instructions or exfiltrate data. Because tool outputs are incorporated into the model’s context, a compromised or malicious server could craft outputs designed to manipulate the model’s subsequent behavior. Defenses include output sanitization, tool output isolation in context, and human-in-the-loop confirmation for sensitive actions.

### 6.3 Authorization

The MCP specification acknowledges that authorization for remote servers requires OAuth 2.0 integration [7]. Local stdio servers rely on OS-level process permissions. Standardized authorization flows remain an active area of development within the MCP ecosystem.

### 6.4 Principle of Least Privilege

MCP servers should be designed to expose only the capabilities required for their intended function. Overly permissive servers that expose broad filesystem access or unrestricted code execution increase the blast radius of a compromised model decision. The specification recommends that hosts implement capability-level access controls and surface sensitive operations for user confirmation.

## 7 COMPARISON WITH RELATED WORK

### 7.1 OpenAI Function Calling and Plugins

OpenAI’s function calling [8] and the now-deprecated ChatGPT Plugins system [9] are the most direct predecessors to MCP. Both enable tool use within OpenAI’s model ecosystem but are tightly coupled to OpenAI’s APIs. MCP differentiates itself through model-agnosticism: any MCP-compliant host can connect to any MCP-compliant server, regardless of the underlying model provider.

### 7.2 LangChain and Tool Abstractions

LangChain [10] and similar orchestration frameworks define their own tool abstractions and integration patterns. While these frameworks are powerful, their integrations are expressed in Python (or JavaScript) code and are framework-specific. MCP’s protocol-level abstraction is language- and framework-agnostic, enabling server implementations in any language and interoperability across ecosystems.

### 7.3 OpenAPI and REST

OpenAPI specifications [11] describe HTTP APIs in a machine-readable format. While LLMs can be given OpenAPI specs to interact with REST services, this requires the model to construct HTTP requests directly—a brittle and error-prone approach. MCP’s tool abstraction provides a higher-level interface with explicit input schemas and richer descriptions tailored for model consumption.

TABLE 1  
MCP Ecosystem Coverage (Early 2026)

Category	Examples	Servers
Developer Tools	GitHub, GitLab, Git	15+
Databases	PostgreSQL, SQLite, MySQL	20+
File Systems	Local FS, Google Drive, S3	10+
Web & Search	Brave, Puppeteer, Fetch	12+
Cloud & Infra	AWS, GCP, Kubernetes	18+
Productivity	Slack, Notion, Linear	25+
Observability	Sentry, Datadog, Grafana	8+
AI & ML	HuggingFace, LangChain	6+

#### 7.4 Semantic Kernel

Microsoft’s Semantic Kernel [12] provides a similar plugin architecture for LLM applications, with support for multiple model backends. MCP and Semantic Kernel address overlapping concerns, and Microsoft has announced support for MCP within Semantic Kernel, suggesting convergence rather than competition between these approaches.

### 8 ECOSYSTEM ADOPTION

Since its release in November 2024, MCP has seen rapid adoption across the AI tooling landscape. Anthropic ships first-party MCP servers for common integrations including filesystems, GitHub, databases (PostgreSQL, SQLite), web search (Brave Search), and developer tools. The open-source community has published hundreds of community servers covering domains such as cloud infrastructure, productivity tools, observability platforms, and scientific data sources.

Host-side support has expanded beyond Claude Desktop to include Cursor, Zed, Sourcegraph Cody, Continue, and various autonomous agent frameworks. The MCP SDK is available in TypeScript, Python, Java, Kotlin, C#, and Go, lowering the barrier to server implementation across technology stacks.

Table 1 summarizes the breadth of official and community integrations as of early 2026.

### 9 OPEN CHALLENGES

#### 9.1 Multi-Agent Coordination

MCP’s current design optimizes for a single host coordinating multiple servers. As AI applications evolve toward networks of collaborating agents, new coordination patterns are needed. The sampling primitive provides a foundation for server-side model invocation, but protocols for agent-to-agent communication, shared context, and distributed task orchestration remain nascent.

#### 9.2 Versioning and Compatibility

Protocol evolution is inevitable. MCP’s initialization handshake supports version negotiation, but the specification does not yet define a comprehensive deprecation policy or backward-compatibility guarantees. As the ecosystem matures, formal versioning semantics will be essential to prevent fragmentation.

#### 9.3 Discoverability

Currently, users must manually configure which MCP servers a host should connect to. There is no standard mechanism for discovering available servers, evaluating their trustworthiness, or composing them automatically based on task requirements. A registry or marketplace model, analogous to package managers or app stores, could address this gap.

#### 9.4 Observability and Debugging

Debugging tool invocations in production LLM applications is challenging. MCP’s protocol layer provides hooks for logging, but standardized observability tooling—distributed tracing, invocation auditing, cost attribution—is not yet part of the specification. This limits operators’ ability to monitor and optimize MCP deployments at scale.

#### 9.5 Formal Verification

The correctness of tool schemas and the safety of tool invocations are currently validated informally. Formal methods for verifying that tool definitions are consistent, that servers behave as specified, and that model decisions remain within authorized boundaries would strengthen the security posture of MCP deployments.

### 10 CONCLUSION

The Model Context Protocol represents a principled and timely response to the integration fragmentation problem facing the AI ecosystem. By drawing on the proven patterns of the Language Server Protocol and grounding its design in widely-adopted standards (JSON-RPC 2.0, OAuth 2.0, JSON Schema), MCP provides a stable foundation for building interoperable AI systems.

Our analysis demonstrates that MCP’s three-primitive architecture—tools, resources, and prompts—cleanly covers the principal modes of AI-tool interaction, while its layered design separates transport concerns from application semantics. The rapid ecosystem adoption observed since the protocol’s release validates its practical utility.

Significant challenges remain. Multi-agent coordination, formal versioning, server discoverability, and production observability are areas where the specification and tooling must mature. Nevertheless, the trajectory of the MCP ecosystem suggests that the protocol is on a path to become foundational infrastructure for the next generation of context-aware AI applications.

## APPENDIX A MCP MESSAGE EXAMPLES

### A.1 Initialization Request

```

1   {
2     "jsonrpc": "2.0",
3     "id": 1,
4     "method": "initialize",
5     "params": {
6       "protocolVersion": "2024-11-05",
7       "capabilities": {
8         "roots": { "listChanged": true },
9         "sampling": {}
      }
    }
  }

```

```

10 },
11   "clientInfo": {
12     "name": "ExampleClient",
13     "version": "1.0.0"
14   }
15 }
16 }
```

Listing 2. MCP initialize request.

- [9] OpenAI, "ChatGPT Plugins," 2023. [Online]. Available: <https://openai.com/blog/chatgpt-plugins>
- [10] H. Chase, "LangChain: Building Applications with LLMs through Composability," 2022. [Online]. Available: <https://www.langchain.com>
- [11] OpenAPI Initiative, "OpenAPI Specification v3.1.0," 2021. [Online]. Available: <https://spec.openapis.org/oas/v3.1.0>
- [12] Microsoft, "Semantic Kernel: An Open-Source SDK," 2023. [Online]. Available: <https://learn.microsoft.com/en-us/semantic-kernel/>

## A.2 Tool Call Request

```

1 {
2   "jsonrpc": "2.0",
3   "id": 2,
4   "method": "tools/call",
5   "params": {
6     "name": "search_web",
7     "arguments": {
8       "query": "Model Context Protocol specification
9         "
10    }
11 }
```

Listing 3. MCP tools/call request.

## APPENDIX B MCP PRIMITIVE SUMMARY

TABLE 2  
Summary of MCP Primitives

Primitive	Controlled By	Purpose
Tools	Model	Execute actions / retrieve data
Resources	Application	Inject context into model
Prompts	User	Reusable interaction templates
Sampling	Server	Request model completions

Furkan Tercan Biography text here.

## ACKNOWLEDGMENTS

The author would like to thank the Anthropic team for open-sourcing the Model Context Protocol specification and SDKs, and the broader open-source community for their rapid development of MCP servers and clients.

## REFERENCES

- [1] Anthropic, "Model Context Protocol Specification," 2024. [Online]. Available: <https://spec.modelcontextprotocol.io>
- [2] Microsoft, "Language Server Protocol Specification," 2016. [Online]. Available: <https://microsoft.github.io/language-server-protocol/>
- [3] OpenAI, "GPT-4 Technical Report," *arXiv preprint arXiv:2303.08774*, 2023.
- [4] Anthropic, "The Claude 3 Model Family: Opus, Sonnet, Haiku," Technical Report, 2024.
- [5] Google DeepMind, "Gemini: A Family of Highly Capable Multi-modal Models," *arXiv preprint arXiv:2312.11805*, 2023.
- [6] JSON-RPC Working Group, "JSON-RPC 2.0 Specification," 2010. [Online]. Available: <https://www.jsonrpc.org/specification>
- [7] D. Hardt, "The OAuth 2.0 Authorization Framework," RFC 6749, IETF, 2012.
- [8] OpenAI, "Function Calling," OpenAI API Documentation, 2023. [Online]. Available: <https://platform.openai.com/docs/guides/function-calling>