Architectural Principles for Multi-Agent Systems Based on The Model Context Protocol

Igor Zuykov

Chief Software Engineer, G-71 Inc
Sugar Hill, GA, USA

Received: 12 Jan 2026 | Received Revised Version: 19 Feb 2026 | Accepted: 17 Mar 2026 | Published: 31 Mar 2026

Volume 08 Issue 03 2026 | Crossref DOI: 10.37547/tajet/Volume08Issue03-12

Abstract

The article examines the architectural role of the Model Context Protocol in the design of multi-agent systems intended to support robust coordination between cognitive and infrastructural components. The relevance of the study is determined by the rapid transition from isolated LLM interactions to distributed agentic environments, in which direct coupling between agents and external APIs, data sources, and tools increases system interdependence, complicates maintenance, and weakens governability. The purpose of the article is to provide a conceptual substantiation of the architectural principles of multi-agent systems based on MCP and to identify their significance for scalability, interoperability, security, and maintainability. The scientific novelty of the study lies in interpreting MCP not merely as an auxiliary integration protocol, but as an autonomous architectural layer that establishes a formalized topology of interaction among hosts, clients, and servers. The principal findings demonstrate that the use of MCP ensures a strict separation of concerns, declarative context management, modular composability, and governance-by-design in capability access. The article concludes that such an approach enhances observability, reduces integration fragility, and creates the foundation for the evolutionary development of multi-agent solutions while preserving control and auditability. The article will be useful for researchers, AI system architects, developers of agentic platforms, and specialists in enterprise digital infrastructure.

Keywords: multi-agent systems, Model Context Protocol, AI architecture, interoperability, modularity, security

© 2026 Igor Zuykov. This work is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0). The authors retain copyright and allow others to share, adapt, or redistribute the work with proper attribution.

Cite This Article: Zuykov, I. (2026). Architectural Principles for Multi-Agent Systems Based on The Model Context Protocol. The American Journal of Engineering and Technology, 8(03), 158–169. https://doi.org/10.37547/tajet/Volume08Issue03-12

Introduction

Large language models have moved from isolated prompt-response interactions toward systems that coordinate planning, retrieval, tool use, and response generation across multiple specialized components (Guo et al., 2024). This shift has amplified a familiar systems problem. With intelligence distributed across agents, architecture and model performance are intertwined. Multi-agent systems must settle on how agents discover their abilities, access data, share context and are kept under control during deployment. Without such rules, scaling almost always leads to brittle architectures, duplicated integrations, and inconsistent security controls (Ferrag et al., 2025).

The main integration challenge occurs at the level of cognition and infrastructure. Current integrations implement agents by wiring them to APIs, databases, search pipelines, and prompt templates. These integrations typically blend business logic, tool invocation logic, and context management within the agent itself. As a result, each new data source increases coupling across the entire system (Xu et al., 2025). Every new integration adds maintenance overhead. Testing becomes narrower than the real workflow. Governance becomes reactive because access paths proliferate faster than they can be audited.

The Model Context Protocol offers a compelling architectural answer because it defines a standardized communication layer between AI applications and the external capabilities on which they depend. Once MCP is treated as an architectural foundation, the discussion shifts toward boundaries, interfaces, orchestration, and capability design. This perspective is especially productive for multi-agent systems, where specialization among agents is sustainable only when the surrounding capability layer is explicit, modular, and controllable.

2. Materials and Methodology

The study of architectural principles for multi-agent systems built on the Model Context Protocol is based on a conceptual and analytical examination of ten academic sources addressing multi-agent architectures, LLM-based agent ecosystems, security risks, and interoperability challenges. The theoretical foundation of the research is formed by survey studies on large language model agents and tool-learning systems, which document the transition from isolated prompt-response models toward distributed agent infrastructures coordinating planning, retrieval, and execution (Guo et al., 2024; Xu et al., 2025). Additional analytical grounding is provided by research on tool-integrated agent frameworks and the structural weaknesses of monolithic agent implementations, where reasoning, execution, and data access are fused into a single operational surface, leading to increased coupling and reduced maintainability (Qu et al., 2024). Subsequent work focused on interoperability and governance in complex digital ecosystems expanded upon the architectural foundations, stressing the importance of standardized interfaces and protocol-level interoperability for sustaining multi-component systems that are designed to scale and coexist (Rosic et al., 2025; Staub et al., 2022). This work is also based on materials from the author of the MCP protocol - Anthropic (Anthropic, 2024; Anthropic, 2025a; Anthropic, 2025b; Anthropic, 2025c).

Our work builds on theoretical work that unifies architectural design with a systematic comparison of agent infrastructure and protocol-mediation capability layers. In particular, a comparative architectural analysis of existing LLM agent implementations shows that LLM agents typically integrate external tools, APIs, and data pipelines directly within their agent logic, leading to fragmented capabilities and uneven governance mechanisms (Xu et al., 2025). The study contrasts these patterns with MCP-mediated architectures, in which hosts, clients, and servers establish an explicit communication topology that separates cognitive processes from infrastructural execution. Conceptual security analysis also specifies the threat model tool-augmented agents are presented with in terms of prompt injection, other capability misuse, and expanded attack surface in multi-agent settings (Ferrag et al., 2025; Zhan et al., 2024). The foundation for this analysis draws upon broader security and governance contexts, namely LLMs and software supply chains. Common themes include bounded exposure of capabilities, protocol-level isolation, and auditable interaction layers (Li & Fung, 2025; Gokkaya et al., 2025). Combined, these methodological perspectives indicate how to derive architectural tenets that achieve stability of distributed reasoning systems while providing interoperability, adaptability, and governance across multi-agent infrastructures.

3. Results and Discussion

MCP is a structured interaction model composed of hosts, clients, and servers. The host is the runtime environment that coordinates tasks and manages the execution context. The client is the protocol implementation inside the host that maintains a stateful connection to a specific MCP server. The server exposes capabilities to the AI system through a standardized interface. This three-layer structure matters because it creates a legible topology for multi-agent systems, where orchestration and capability access can evolve without collapsing into a single monolithic agent loop.

The protocol organizes server functionality through three primitives. Tools provide executable functions that can act on external systems. Resources provide structured or unstructured data that can be retrieved as context.

Prompts provide reusable templates that encapsulate recurring interaction patterns. These primitives are sufficiently abstract to support heterogeneous environments and sufficiently concrete to stabilize implementation practice. They also make capability exposure inspectable, which is essential when agents operate in domains that require strong security and auditability.

A key consequence of this design is that the AI application does not interact directly with tools or data sources. It communicates with MCP servers through clients that mediate discovery, negotiation, and routing. This indirection does more than simplify integration. It establishes an architectural boundary that keeps reasoning systems conceptually distinct from external execution systems. In a multi-agent setting, that separation allows specialized agents to share infrastructure without internalizing every detail of the environment they inhabit. The MCP architecture concept is shown in Figure 1.

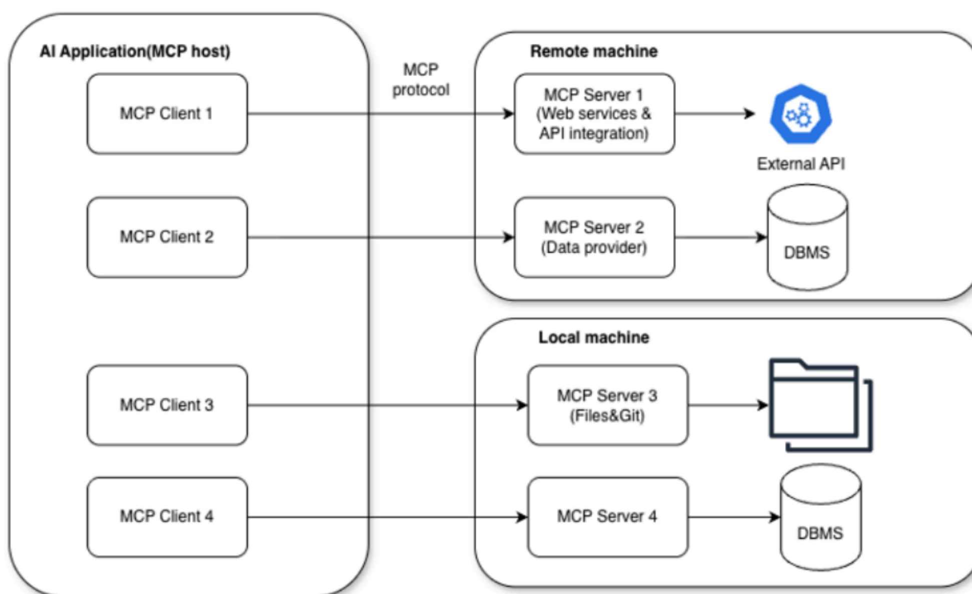

Fig. 1. MCP architecture

A robust multi-agent system depends on more than the existence of several agents with distinct labels. The architecture must determine how those agents relate to tools, data, prompts, and each other at scale. In the MCP-based view, four principles provide this structure. They are strict separation of concerns, security and governance by design, modular composability and interoperability, and declarative context provisioning. Together, they establish the structural logic that enables intelligence to be distributed while keeping control points visible and manageable.

These principles emerge from the standardized interaction layer introduced by MCP. Once capabilities are exposed through protocol-defined servers, the architecture can distinguish cognitive roles from execution roles, replace point-to-point integration with reusable interfaces, and move context management into a managed infrastructure layer. This has direct implications for maintainability, testing, policy enforcement, and the long-term evolution of the system. It also reduces the amount of hidden glue code that often accumulates around agent frameworks.

The four principles should be read as mutually reinforcing. Separation of concerns is strengthened by capability isolation. Security becomes more tractable when exposure is narrow and explicit. Composability becomes practical when context and prompts are standardized assets. Context provisioning becomes reliable when servers and interfaces are versioned. The remaining subsections examine each principle in detail.

A recurring weakness in monolithic agent implementations is the fusion of planning, reasoning, data retrieval, and action execution within a single agent boundary (Qu et al., 2024). That fusion tends to produce low cohesion because the same component is asked to interpret goals, call tools, manage failures, and synthesize outputs. The resulting system often becomes difficult to extend because a change in one tool or dataset propagates through the agent logic. In multi-agent

systems, this weakness scales quickly, since each new specialist agent can reproduce the same pattern of entanglement.

MCP introduces a clear agent-environment boundary. The agent, operating within the host, is responsible for cognitive tasks such as goal decomposition, planning, and integrating information across sources. Execution and data access are delegated to MCP servers that expose narrowly scoped capabilities through resources, tools, and prompts. The host coordinates communication across these components. This yields a disciplined division of labor in which cognitive specialization and infrastructural specialization can be optimized separately.

The architectural consequence is independent evolution. A reasoning engine can be improved without rewriting tool implementations. A server maintainer can revise retrieval logic or enforcement policies without forcing changes to the agent's internal reasoning stack, provided that the protocol contract remains stable. This property is especially important in enterprise settings, where architectural longevity depends on minimizing cross-component coupling while preserving room for incremental upgrades. Figure 2 illustrates the MCP based Multi-agent system.

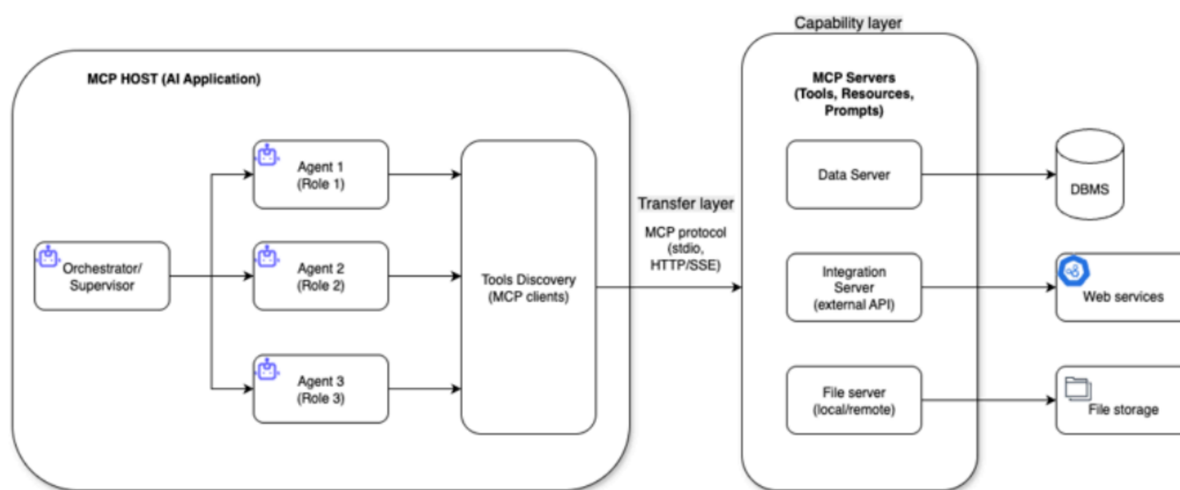

Fig. 2. MCP based Multi-agent system

Security in multi-agent systems cannot be treated as an add-on because agents are inherently exposed to adversarial inputs, ambiguous instructions, and heterogeneous tools (Li & Fung, 2025). Direct agent access to external systems increases the risk of injection attacks, unintended execution, data leakage, and privilege escalation (Zhan et al., 2024). When multiple agents coexist, the attack surface expands further because the system includes more routes through which malicious or malformed inputs can propagate. A secure architecture requires explicit constraints at the capability layer.

MCP embeds the principle of least privilege into the architecture. Agents do not carry inherent access to external systems. They can only interact with capabilities that a connected server explicitly exposes. Authentication, authorization, rate limiting, and input sanitization are enforced at the server layer. In addition, host-mediated isolation constrains what servers can observe and how they can interact. A server does not gain access to the full conversation history. Servers do not directly interact with one another. Capability exposure remains bounded and inspectable.

This design relocates governance to a declarative, auditable layer. Access policies become properties of server configuration and protocol exposure. The result is a smaller blast radius for compromised components and a clearer basis for auditing and incident response. In a multi-agent system, such a design supports organizational trust because security policies are applied where capabilities are exposed, where logs can be captured, and where enforcement can be verified systematically. Security boundaries are shown in Figure 3.

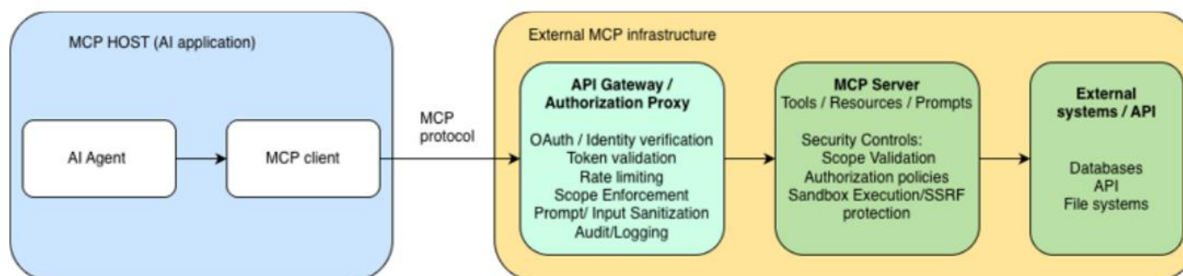

Fig. 3. Security Boundaries

Agent ecosystems frequently drift toward fragmentation (Staub et al., 2022). Tools created for one framework often remain tightly bound to that framework's abstractions, packaging assumptions, and runtime model. This leads to duplication of effort because the same external capability is reimplemented many times for different agent stacks (Rosic et al., 2025). It also limits reuse, as teams become dependent on local conventions. In a multi-agent architecture, fragmentation undermines the very efficiencies that specialization is supposed to create.

MCP addresses this problem by treating servers as framework-agnostic capability providers. Any agent that conforms to the protocol can use a compatible server regardless of the framework in which it was built. A database server, a web search server, or a file server can therefore serve multiple agents and multiple applications. This creates the conditions for plug-and-play interoperability across an evolving ecosystem of agents and services. The architecture begins to resemble a shared capability layer.

The practical effect is a reusable infrastructure library that can be audited, improved, and shared across teams. Multi-agent systems are composed of specialized reasoning units that operate over common services. Development time decreases because integrations are built once and reused. Architectural complexity becomes more legible because the same capability layer can support several workflows without hidden rewiring. In environments where many teams experiment with agentic applications simultaneously, this form of modularity can determine whether the ecosystem grows coherently or drifts into operational sprawl. Figure 4 illustrates the multiple agents sharing a common set of MCP servers.

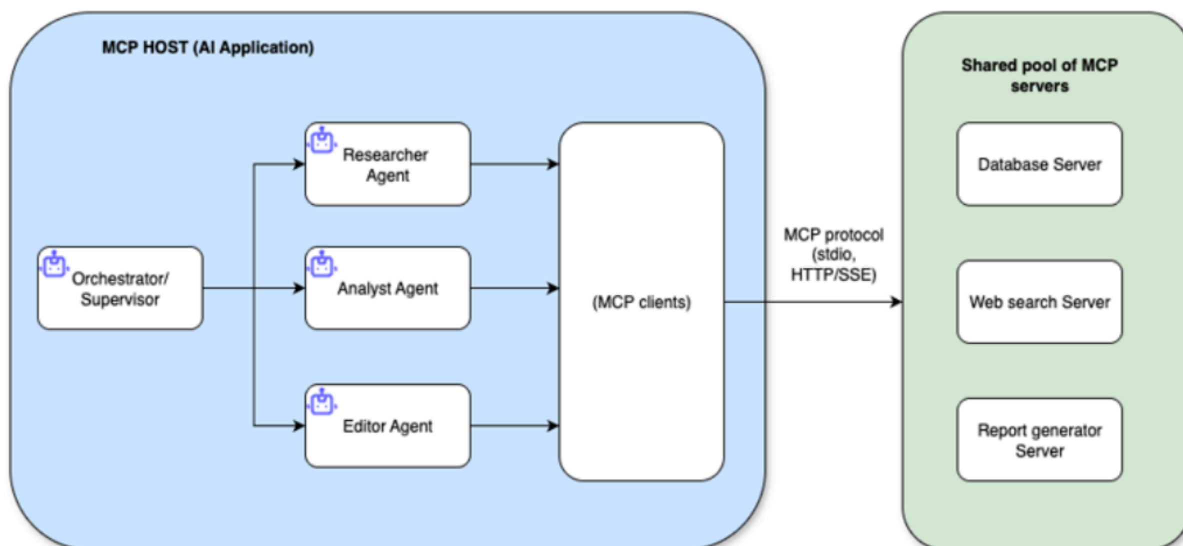

Fig. 4. Multiple Agents Sharing a Common Set of MCP Servers

Context management remains one of the least stable parts of many LLM systems (Arslan et al., 2024). Developers manually chunk files, define retrieval heuristics, embed documents into vector pipelines, and maintain prompt strings inside application code. These solutions can work in prototypes, yet they become brittle when systems

require version control, coordinated updates, discoverability, and cross-agent reuse. In multi-agent settings, the problem intensifies because different agents may need distinct views of shared context while still relying on common governance and provenance controls.

MCP introduces a more declarative model through resources and prompts. Agents request context using structured identifiers such as a resource URI or a prompt name. They do not need to know how the underlying retrieval, chunking, embedding, formatting, or template management is implemented. Those concerns are handled by the server. This shifts context orchestration out of agent logic and into a managed infrastructure layer, where updates can be made centrally and consumed consistently by many agents.

The architectural gain is substantial. Context becomes a first-class asset that is discoverable, versioned, and administratively controlled. That enables prompt versioning, context caching, retrieval-augmented generation pipelines, and controlled experiments with alternative templates or retrieval policies. For multi-agent systems, it also reduces semantic drift because agents can depend on shared context contracts. The result is a platform where context is treated with the same seriousness as APIs and data schemas.

MCP and multi-agent architecture share a common structural logic. The protocol's internal logic already assumes a structured separation between orchestration, capability access, and capability provision. This maps naturally onto multi-agent systems, where coordination must coexist with specialization. The host becomes the orchestration locus. Clients maintain protocol-specific connections. Servers become externally governed capability providers. Under this arrangement, the architecture supports distributed cognition without surrendering infrastructural clarity.

The lifecycle framing presented in the source material clarifies this further by linking MCP design to phases of creation, operation, and update. During creation, servers are designed around specific capabilities, which aligns with the principle of separation of concerns. During operation, servers remain isolated and composable, which supports both modularity and security. During updates, standardized and versioned interfaces allow agents and servers to evolve independently. This lifecycle logic transforms protocol compliance into a durable architectural discipline. The following table captures this mapping.

Table 1. Lifecycle-Based Architecture of MCP

Lifecycle Phase	MCP Design	Multi-Agent Principle	Practical Impact
Creation Phase	Servers focus on specific capabilities	Separation of Concerns	Agents reason. Servers act. Teams can develop and maintain each independently.
Operation Phase	Servers are isolated and composable	Modular Composability	Agents can mix and match servers. Teams can build a shared, audited capability library.
Operation Phase	Isolated connections and limited context	Security and Governance by Design	Least privilege becomes a default condition. The architecture mitigates tool conflicts and limits blast radius from a compromised server.
Update Phase	Standardized, versioned interface	Extensibility and Maintainability	Servers and agents can be updated independently. System evolution avoids disruptive full-scale upgrades.

This lifecycle table is valuable because it expresses a deeper systems insight. Architectural quality in multi-agent systems depends on whether change can occur without re-entangling cognition, execution, security, and context. MCP's design supports that quality by assigning each concern a stable structural location. In practical terms, this means that a growing system is more likely to

remain intelligible as its number of agents, tools, and workflows increases.

To clarify the practical significance of these architectural principles, it is useful to move from the level of abstract design logic to that of an applied system scenario. A concrete example makes it possible to observe how separation of concerns, governance of capability access, modular composability, and declarative context provisioning operate not merely as theoretical qualities, but as organizing mechanisms within an end-to-end workflow. Examining such a scenario therefore helps translate the preceding architectural analysis into an operational model that demonstrates how MCP can structure coordination among specialized agents and shared infrastructural services.

A customer support assistant offers a concrete illustration of how MCP principles can be translated into a multi-agent system. In this architecture, the host serves as the runtime platform, providing orchestration and state management logic. Within it reside three MCP clients implemented as specialist agents. The Classifier Agent determines intent and extracts key entities. The Retriever Agent gathers customer-specific data and relevant knowledge articles. The Reply Generator Agent prepares a personalized draft response. This division of labor keeps each agent narrowly focused while allowing the host to retain workflow control. The customer support assistance workflow is shown in Figure 5.

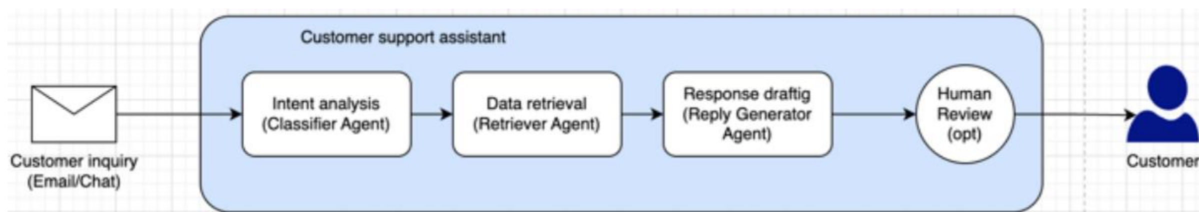

Fig. 5. Customer support assistance workflow

The agent roles can be represented as follows (see table 2).

Table 2. The agent roles

Agent	Role
Classifier Agent	Analyzes the incoming message to determine intent, such as billing or technical support, and extracts key entities such as customer ID or product.
Retriever Agent	Gathers customer-specific data and relevant knowledge base articles based on the detected intent and extracted entities.
Reply Generator Agent	Creates a personalized preliminary response, including relevant links and explanations.

The shared capability layer contains the servers that mediate access to external systems. These servers enforce authorization policies and expose minimal, well-defined capabilities through resources, tools, and prompts. The architecture described in the source identifies at least CRM, Ticketing, Billing, Knowledge Base, and Response Template servers as distinct providers. Their responsibilities are sufficiently bounded to support auditability and reuse across multiple agents or support workflows. This arrangement makes the capability layer reusable and keeps domain authority located in the servers that own the relevant external interactions. The server capability model can be summarized as follows (see table 3).

Table 3. Functional Distribution of MCP Servers

MCP Server	Capabilities Provided
CRM Server	Resources: customer profile, purchase history, support tickets. Tools: lookup customer by ID, retrieve account details with authorization, retrieve tickets.
Ticketing Server	Manages the lifecycle of customer support cases and acts as the system of record for support interactions. Resources: support tickets, ticket history.
Billing Server	Provides access to financial transactions, subscriptions, invoices, and refunds. Resources: customer subscriptions, payment transactions, invoices, refund records.
Knowledge Base Server	Resources: articles, FAQs, troubleshooting guides. Tools: search articles by keyword, retrieve article content by ID.
Response Template Server	Prompts: versioned templates for different intents such as prompt://support/billing. Resources: style guides, policy documents.

This reference architecture demonstrates the cumulative force of the four principles. Each agent has a single responsibility. Capability access is policy-governed at the server layer. Servers can be shared across agents and even across teams. Prompts and contextual assets are fetched from managed services. The result is a support workflow whose structure remains explicit, whose components are independently testable, and whose maintenance burden is distributed across clear architectural boundaries. The reference architecture is shown in Figure 6.

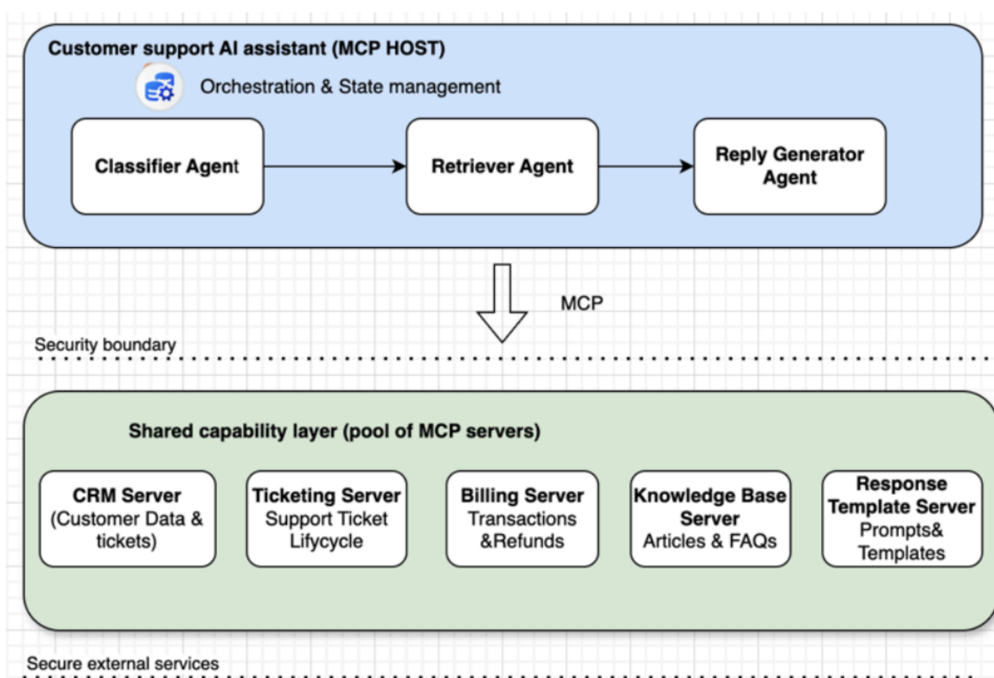

Fig. 6. Reference Architecture

This end-to-end process begins with the customer inquiry being entered into the support system. The host will receive the message and run the Classifier Agent against the original message text. For example, the classifier receives a classification prompt from the Response Template Server, analyzes the message, and returns a relative response with an intent label and a number of identifiers extracted. These labels and identifiers are pre-defined based on the nature of a specific intent or scenario. In this example, when the user complaints about a double charge, the classifier returns a billing-related intent and a customer identifier. This first step shows how prompts can function as managed workflow assets within a multi-agent pipeline.

The host then invokes the Retriever Agent using the classification result. The retriever calls the CRM Server to obtain profile information, subscription details, and recent transactions. It also queries the Knowledge Base Server for articles related to the duplicate-charge issue and returns the combined result to the host. At this stage, the architecture demonstrates the separation between inference and capability access. The retriever does not embed domain ownership. It orchestrates server-mediated retrieval under explicit authorization handled at the server layer.

In the final stage, the host passes the gathered customer data, the retrieved knowledge articles, and the identified intent to the Reply Generator Agent. That agent fetches a billing response template from the Response Template Server, injects customer-specific details, and produces a draft response. The workflow includes a human review step before sending, which is especially important in support contexts involving billing, policy interpretation, or reputational risk. This closing stage reveals an important property of MCP-based orchestration. Human supervision can be inserted at controlled points in the workflow without modifying the agent architecture or embedding manual exceptions into every agent.

An MCP-driven multi-agent system yields operational advantages because every interaction with an external capability is expressed through an explicit protocol call. This improves observability. Requests can be traced through the host, the client connection, and the target server. Debugging becomes more methodical because developers can inspect the orchestration chain and the usage of capabilities. This also serves to improve auditability, since both capability exposure and invocation use the same formal interface.

Resilience is improved because capabilities are distributed over independent servers, allowing their components to be replaced, updated, and scaled independently. Upgrading one server does not require the agent layer to be rebuilt as long as the interface is compatible with the previous version. Likewise, the reasoning stack can evolve without constant re-engineering of every external integration. This decoupled evolution is central to long-lived systems, where operational stability depends on the ability to change one subsystem while preserving the behavior of others.

A further advantage lies in the ecosystem's growth. Organizations can gradually assemble an internal catalog of trusted MCP servers that functions as a reusable capability marketplace. New agentic applications can then be developed by composing existing services. This supports institutional learning. It also encourages convergence on vetted interfaces, audited services, and reusable prompt assets. Over time, the architecture can therefore generate compounding returns in development velocity, governance consistency, and infrastructural coherence.

Despite its promise, the MCP ecosystem remains at an early stage of maturity. The architecture described in the source material already identifies several unresolved concerns. Centralized security oversight is limited. Multi-tenant environments remain vulnerable to configuration drift. Strong authentication and authorization frameworks across clients and servers require further consolidation. These issues do not negate the architectural value of MCP. They indicate that the protocol still needs surrounding standards, tooling, and operational practice to support large-scale deployment.

A second set of challenges concerns server distribution and lifecycle risk. Even in the decentralized version, a number of threats remain, such as installer spoofing, unofficial packages, or redeploying an outdated or compromised version of the server. The problem becomes more complicated in multi-agent systems, since the capability layer is often a shared dependency (Gokkaya et al., 2025). In addition, whether powered by a single server or by a distribution pipeline, this type of indexing has cascading operational effects across different agents and use cases that raise questions of signing, provenance, trusted catalogs, and continuous security validation.

The support-assistant example also helps summarize the practical meaning of these open questions by showing

what each principle must achieve under real conditions, as shown in Table 4.

Table 4. Applied Principles in the Support-Assistant Architecture

Principle	Demonstrated By
Separation of Concerns	Each agent has a single responsibility. The host handles coordination. Agents handle domain tasks.
Security by Design	The CRM Server enforces access control. Agents never see raw credentials. A compromised Knowledge Base Server cannot access customer data or conversation history. All access is logged at the server level.
Modular Composability	The same CRM Server could support other agents such as an analytics agent. The Response Template Server is a shared asset usable across different support teams.
Declarative Context	Prompts for classification and drafting are fetched from a server. They can be updated without redeploying agents.

The open research problem concerns governance at scale. Future work will need to clarify how trusted server ecosystems are curated, how policy enforcement is standardized, how versioning interacts with safety guarantees, and how organizations can preserve the modular benefits of MCP while tightening control over distribution and configuration.

4. Conclusion

The analysis presented in this article supports the conclusion that the Model Context Protocol constitutes a consequential architectural foundation for multi-agent systems precisely because it formalizes the interface between reasoning processes and external capabilities. As large language model applications evolve beyond isolated prompt-response exchanges toward distributed arrangements of planning, retrieval, tool use, and synthesis, architectural discipline becomes inseparable from system reliability. Within this context, MCP acquires significance through its structured topology of hosts, clients, and servers, which renders the operational surface of a multi-agent system legible, governable, and extensible. This protocol-defined mediation layer reduces the infrastructural opacity that often accompanies direct agent coupling with APIs, databases, and retrieval pipelines, thereby establishing the conditions under which specialization among agents can persist without generating cumulative fragility.

The findings further indicate that the principal architectural value of MCP lies in the four mutually reinforcing principles derived from its design logic. Strict separation of concerns allows cognitive functions such as planning, decomposition, and synthesis to remain distinct from execution and data access. Security and governance by design relocate access control, authentication, authorization, and auditability to the server layer, where capability exposure remains narrow and inspectable. Modular composability and interoperability enable shared capability providers to serve multiple agents and applications across heterogeneous frameworks. Declarative context provisioning transforms prompts and resources into managed infrastructural assets that can be discovered, versioned, and reused across workflows. Taken together, these principles define an architectural structure capable of sustaining distributed intelligence while preserving clarity of control, maintainability, and policy enforcement.

The lifecycle interpretation developed in the discussion deepens this conclusion by showing that MCP does more than standardize communication. It stabilizes system evolution across creation, operation, and update phases.

During creation, capability-specific server design reinforces infrastructural specialization. During operation, isolated and composable servers support shared reuse, bounded exposure, and more coherent governance. During updates, standardized and versioned interfaces permit agents and servers to evolve independently, which is indispensable for long-lived enterprise systems. The customer support reference architecture substantiates these claims at the application level. Its division into classifier, retriever, and reply generator agents demonstrates how narrowly scoped cognitive roles can be coordinated by a host while relying on a shared capability layer composed of CRM, ticketing, billing, knowledge base, and response template servers. This arrangement yields explicit workflow control, independently testable components, improved observability, and the ability to insert human review at critical decision points without disrupting the architecture.

At the same time, the article demonstrates that the promise of MCP-based multi-agent systems remains accompanied by unresolved governance and security questions. The protocol already provides an architecture that enables least privilege, bounded context exposure, and auditable access to capabilities. However, the broader ecosystem still faces immaturity in centralized oversight, multi-tenant configuration discipline, strong cross-component authentication, trusted distribution, provenance assurance, and lifecycle validation of server implementations. These limitations do not diminish the conceptual strength of the architectural model. They identify the frontier on which future research and operational standardization must concentrate. On this basis, the article justifiably concludes that MCP should be understood as a durable architectural foundation for multi-agent systems, one that enables scalable specialization, reusable capability layers, and more intelligible system growth, while simultaneously opening an important research agenda around governance at scale.

References

1. Anthropic. (2024). *Introducing the Model Context Protocol*. Anthropic. https://www.anthropic.com/news/model-context-protocol
2. Anthropic. (2025a). *Architecture Overview - Model Context Protocol*. https://modelcontextprotocol.io/docs/learn/architecture
3. Anthropic. (2025b). *Introduction - Model Context Protocol*. Anthropic. https://modelcontextprotocol.io/
4. Anthropic. (2025c). *Security best practices - Model context protocol*. Anthropic. https://modelcontextprotocol.io/docs/tutorials/security/security_best_practices
5. Arslan, M., Ghanem, H., Munawar, S., & Cruz, C. (2024). A Survey on RAG with LLMs. *28th International Conference on Knowledge Based and Intelligent Information and Engineering Systems (KES 2024)*, *246*, 3781–3790. https://doi.org/10.1016/j.procs.2024.09.178
6. Ferrag, M. A., Tihanyi, N., Hamouda, D., Maglaras, L., Lakas, A., & Debbah, M. (2025). From prompt injections to protocol exploits: Threats in LLM-powered AI agents workflows. *ICT Express*, *12*(2), 353–383. https://doi.org/10.1016/j.icte.2025.12.001
7. Gokkaya, B., Aniello, L., & Halak, B. (2025). Software supply chain: A taxonomy of attacks, mitigations and risk assessment strategies. *Journal of Information Security and Applications*, *97*, 104324. https://doi.org/10.1016/j.jisa.2025.104324
8. Guo, T., Chen, X., Wang, Y., Chang, R., Pei, S., Chawla, N. V., Wiest, O., & Zhang, X. (2024). Large Language Model Based Multi-agents: A Survey of Progress and Challenges. *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence Survey Track*, 8048–8057. https://doi.org/10.24963/ijcai.2024/890
9. Li, M. Q., & Fung, B. C. M. (2025). Security concerns for Large Language Models: A survey. *Journal of Information Security and Applications*, *95*, 104284. https://doi.org/10.1016/j.jisa.2025.104284
10. Qu, C., Dai, S., Wei, X., Cai, H., Wang, S., Yin, D., Xu, J., & Wen, J.-R. (2024). Tool Learning with Large Language Models: A Survey. *Frontiers of Computer Science*, *19*, 198343. https://doi.org/10.1007/s11704-024-40678-2
11. Rosic, M., Sumic, D., & Males, L. (2025). Semantic Interoperability of Multi-Agent Systems in Autonomous Maritime Domains. *Electronics*, *14*(13), 2630. https://doi.org/10.3390/electronics14132630
12. Staub, N., Haki, K., Aier, S., & Winter, R. (2022). Governance Mechanisms in Digital Platform Ecosystems: Addressing the Generativity-Control Tension. *Communications of the Association for*

Information Systems, *51*(1). https://doi.org/10.17705/1CAIS.05137

13. Xu, W., Huang, C., Gao, S., & Shang, S. (2025). LLM-Based Agents for Tool Learning: A Survey. *Data Science and Engineering*, *10*, 533–563. https://doi.org/10.1007/s41019-025-00296-9

14. Zhan, Q., Liang, Z., Ying, Z., & Kang, D. (2024). InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated Large Language Model Agents. *Findings of the Association for Computational Linguistics: ACL 2022*, 10471–10506. https://doi.org/10.18653/v1/2024.findings-acl.624