Journal of Business Intelligence and Data Analytics

Journal homepage: www.sciforce.org
ISSN : 2998-3541

Open Access **Research Article**

Architecting MCP-Based Platforms for Enterprise-Scale Agentic Generative AI

Karthik Perikala*

Senior Principal Software Engineer, The Home Depot., United States

Abstract

Enterprise adoption of generative AI is rapidly shifting from isolated prompt-driven applications toward complex agentic systems that integrate retrieval, reasoning, and tool execution. As these systems grow in scale, the lack of a standardized interaction model between agents and external capabilities introduces challenges in reliability, observability, security, and operational governance.

This paper presents aplat form architecture centered on the *Model Context Protocol* (MCP) as a first-class systems abstraction for enterprise-scale agentic generative AI. MCP servers act as strongly isolated, capability-oriented services that expose tools, data access, and actions to agents throughwell-defined contracts. This separation enables controlled tool invocation, bounded execution, and fault isolation across complex multi-agent workflows.

We describe the architectural principles, execution lifecycle, and operational characteristics of

MCP-based platforms, including agent orchestration, context management, latency governance, and failure containment. The paper draws on production deployment experience and provides guidance for building scalable, cost-aware,and reliable agentic AI systems in enterprise environments.

Keywords: Model Context Protocol, Agentic AI, Generative AI Platforms, Distributed Systems, Enterprise Architecture

Introduction

Generative AI systems are undergoing a fundamental architectural transition. Early deployments focused primarily on single-turn prompt completion, where language models operated as isolated inference engines. In contrast, modern enterprise use cases increasingly require *agentic behavior*: multi-step reasoning, interaction with external tools, retrieval of domain knowledge, and execution of business actions.

This transition introduces new systems-level challenges.Agent workflows may span multiple language model invocations, tool calls, and data sources, often under strict latency, cost, and reliability constraints. Without clear execution boundaries,thesesystemsriskunboundedfan-out, cascading failures, and unpredictable operational behavior.

The Model Context Protocol (MCP) addresses these challenges by formalizing how agents interact with external capabilities. Rather than embedding toollogicdirectlywithinagents, MCPexternalizes tools into independently deployable servers with explicit schemas, execution semantics, and lifecycle management. This decoupling enables stronger isolation, observability, and governance while preserving flexibility at the agent layer.

This paper examines MCP not as anapplication-level convenience, but as aplatform-level abstraction analogous to microservices in distributed systems.We argue

that MCP-based architectures provide a scalable foundation for enterprise agentic AI, enabling controlled execution, predictable tail latency, and sustainable operational cost as generative AI systems evolve in complexity.

MCP Platform Architecture

Architectural Overview

The MCP platform establishes a strict separation between agent reasoning and external capability execution. Agents focus exclusively on planning, decomposition, and decision-making, while all side-effecting operations are delegated to MCP servers deployed as independent services.

This separation is foundational to achieving scalability and operational safety in agent-based systems. By preventing agents from directly invoking infrastructure, databases, or external APIs, the platform enforces deterministic execution boundaries and reduces the impact scope of failures.

From Monolithic Agents to MCP Servers Early agent implementations typically embedded prompts, tools, data access, and business logic within a single monolithic application. As the number of tools and workflows grew, this approach made versioning, testing, and operational governance increasingly difficult.

MCP externalizes execution into independently deployable servers with explicit schemas and execution contracts. This decoupling enables controlled evolution of tools, safer experimentation, and clearer ownership across large engineering organizations.

Received date: October 11 2025 **Accepted date:** October 17, 2025; **Published date:** November 05 2025

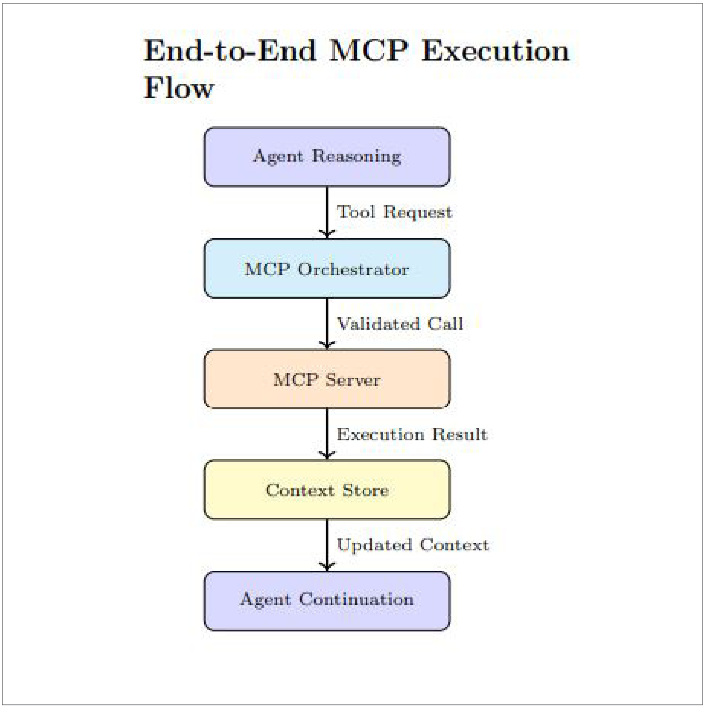
Corresponding Author:* *Perikala, K, Senior Principal Software Engineer, The Home Depot., United States;* E- mail: karthik.perikala2512@gmail.com

Core Platform Components

The MCP platform is composed of four primary components:

- **Agents**: Reasoning entities responsible for planning and tool selection
- **MCP Servers**: Capability-specific, stateless execution services
- **Orchestrator**: Routing, policy enforcement, retries, and concurrency control
- **Context Store:** Structured state for execution continuity

Each component scales independently, enabling fine-grained capacity management and fault isolation as agent complexity and workload volume increase.



Execution Semantics

Each transition in the execution flow represents a strict validation boundary. The orchestrator validates tool schemas, enforces authorization policies, and applies rate limits before forwarding requests to MCP servers.

MCP servers execute deterministically and return structured results that are persisted to the context store. This ensures downstream reasoning operates on stable, versioned inputs and enables safe retries or partial re-execution.

Explicit Context Management Materializing context between execution steps enables auditable execution traces, deterministic replay, and partial workflow recovery. This design supports pause-resume semantics and prevents loss of conversational or execution state during retries or failures.

Single-System MCP Execution Flow

Search Execution Path

To illustrate the MCP execution model concretely, this section focuses on a single domain capability: Search. The same interaction pattern applies uniformly to other MCP-backed domains such as pricing, inventory, recommendations, fulfillment, and assortment.

The execution flow begins at the UI, where user intent is forwarded to a Workflow Agent responsible for high-level intent decomposition. Rather than embedding business logic or system access directly, the Workflow Agent delegates domain-specific reasoning to a specialized Search Agent.

The Search Agent performs semantic interpretation, ranking strategy selection, constraint resolution, and fallback planning. Once an execution plan is formed, the agent invokes the Search MCP server using a strongly typed tool interface, ensuring schema validation and bounded execution semantics.

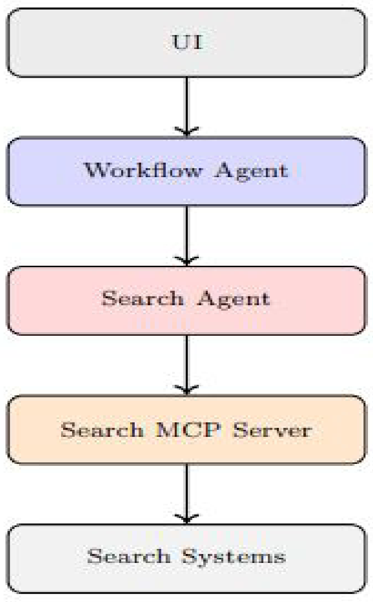


Figure 2: MCP execution flow for a single domain capability (Search).

Generalization Across Domains

This execution pattern generalizes across all enterprise domains. Each domain introduces a specialized Agent and MCP Server pair without modifying the Workflow Agent or orchestration semantics. This preserves architectural consistency while enabling independent evolution of domain capabilities.

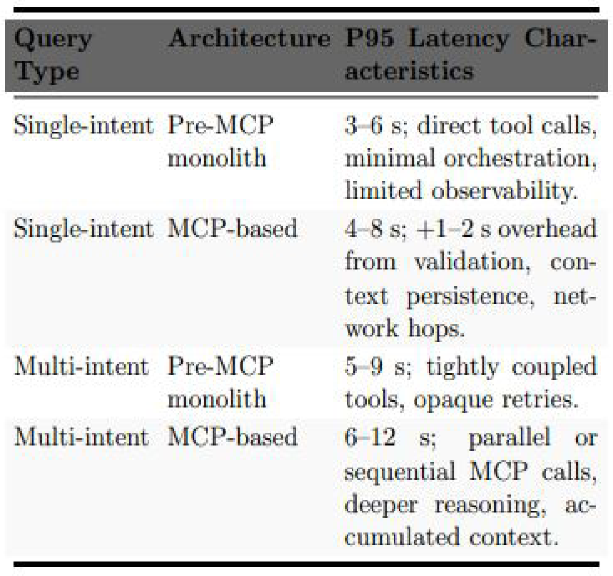
MCP Server Responsibility

The Search MCP server serves as the exclusive execution gateway to downstream search infrastructure. It enforces schema validation, authorization, rate limits, and request normalization before interacting with enterprise systems.

The MCP server contains no business reasoning and produces deterministic, side-effect-free responses. This separation ensures predictable execution while allowing MCP servers to remain stateless, horizontally scalable, and reusable across agents and workflows.

Latency and Throughput Benchmarks

Table summarizes observed P95 latency under a representative workload of 10 TPS, average 10,000-token prompts, and up to eight tools exposed per MCP server.



Query Type	Architecture	P95 Latency Characteristics
Single-intent	Pre-MCP monolith	3–6 s; direct tool calls, minimal orchestration, limited observability.
Single-intent	MCP-based	4–8 s; +1–2 s overhead from validation, context persistence, network hops.
Multi-intent	Pre-MCP monolith	5–9 s; tightly coupled tools, opaque retries.
Multi-intent	MCP-based	6–12 s; parallel or sequential MCP calls, deeper reasoning, accumulated context.

Table 1: P95 latency comparison across single- and multi-intent queries.

Interpretation

Introducing MCP servers increases end-to-end latency by approximately 1–2 seconds due to explicit orchestration, schema validation, authorization, and context materialization. This overhead is an intentional trade-off for improved observability, fault isolation, and deterministic execution.

Latency is dominated by agent reasoning depth and the number of MCP tool invocations rather than raw LLM inference alone. Multi-intent workflows may execute MCP calls sequentially or in parallel depending on dependency structure, yet observed P95 bounds remain suitable for interactive enterprise applications.

Context Management Model

Explicit Context Materialization

A defining characteristic of the MCP platform is the explicit materialization of execution context between reasoning and execution steps. Rather than relying on implicit conversational state embedded within application runtimes, all intermediate inputs, outputs, and execution metadata are persisted as structured context objects exchanged across MCP boundaries.

It is important to note that prior monolithic agent systems already achieved high levels of observability through OpenTelemetry, structured logging, and agent tracing frameworks such as LangSmith. MCP does not replace these capabilities, nor does it fundamentally alter the mechanics of tracing or metrics collection.

The key distinction lies not in observability depth, but in context ownership. Under MCP, context is no longer an internal implementation detail of a single application. Instead, it becomes an explicit, portable artifact that can be consumed, replayed, and reasoned about across independently deployed MCP servers and agent runtimes.

Deterministic Replay and Workflow Contro

Explicit context exchange enables deterministic replay of agent workflows at the protocol boundary. Given the same user intent and persisted context, agent reasoning can be re-executed without reissuing side-effecting calls to external systems. This capability is especially valuable when workflows span multiple domain capabilities owned by different teams. Failures can be isolated to individual steps, allowing targeted retries or alternative execution paths without resetting the entire workflow or recomputing upstream reasoning.

Isolation of Reasoning and Execution State

By externalizing execution context, MCP isolates agent reasoning state from tool execution infrastructure. Agents remain stateless across invocations, while MCP servers operate as deterministic execution endpoints.

This separation eliminates hidden coupling between prompt logic, tool behavior, and runtime state that commonly emerges in monolithic agent applications as prompt complexity and tool surface area grow.

Tool Contracts, Prompts, and MCP APIs

Domain-Specific MCP Servers

The primary architectural benefit of MCP lies in the creation of domain-specific MCP servers that encapsulate tools, resources, and prompts for a well-defined business capability. Examples include search, pricing, inventory, fulfillment, or recommendations.

Each MCP server acts as a canonical owner of its domain contracts. Tools, prompt templates, and resource definitions are versioned, tested, and evolved independently from agent applications. This contrasts with monolithic designs, where prompts and tools are tightly coupled to application releases and difficult to evolve safely.

Externalization and Versioning of Prompts

Prompts are treated as first-class artifacts within MCP servers rather than inline strings embedded in application code. This externalization enables explicit versioning, controlled rollout, and backward compatibility across agents and consuming applications.

Applications may query MCP servers to retrieve prompt templates dynamically or invoke tools directly without embedding prompt logic locally. This pattern allows multiple applications to share the same domain intelligence while maintaining independent release cycles.

Tool Invocation Across Applications

MCP servers expose tools and prompts designed specifically for consumption by generative AI agents and AI-driven services. This design enables consistent reuse of domain intelligence across conversational agents, retrieval-augmented generation pipelines, and multi-agent workflows, while ensuring that all generative interactions follow the same validated execution and governance model.

By consolidating domain logic into MCP servers, enterprises avoid duplicating tool implementations, reduce prompt drift, and enforce consistent execution semantics across heterogeneous consumers.

Design Implication The MCP model shifts agent architectures away from monolithic prompt-and-tool bundles toward a modular, service-oriented ecosystem. The primary gains are not improved observability, but improved governance, reuse, and evolvability of prompts, tools, and domain intelligence at enterprise scale.

Latency Characteristics

End-to-End Latency Decomposition

End-to-end latency in MCP-based systems is composed of four dominant factors: agent reasoning time, orchestration overhead, MCP server execution, and downstream system response latency. These components are explicitly separated to enable targeted optimization without cross-layer interference.

Agent reasoning latency is driven primarily by prompt size, reasoning depth, and conversation context length. Orchestration overhead remains minimal, consisting of schema validation, routing, and policy checks. MCP server latency is largely determined by downstream system behavior and is isolated per domain, preventing unrelated capabilities from influencing one another.

Observed Latency Trade-offs

Compared to pre-MCP monolithic designs, MCP-based execution introduces an additional 1–2 seconds of overhead at the P95 level. This increase is primarily attributable to explicit orchestration, context serialization, and network boundaries between agents and MCP servers. In exchange, latency variance is significantly reduced. Tail behavior becomes predictable and bounded, which is critical for enterprise interactive systems operating under mixed workloads and partial dependency degradation.

Throughput and Concurrency

Horizontal Scaling Behavior

MCP servers are stateless and scale horizontally with minimal coordination. Throughput increases linearly with replica count, allowing capacity to be provisioned independently for each domain such as search, pricing, or inventory.

This domain-level scaling avoids coarse-grained over-provisioning and enables cost-efficient handling of asymmetric traffic patterns common in enterprise workloads.

Concurrency Management

Concurrency limits are enforced per MCP server to protect downstream systems. Requests beyond configured thresholds are queued or throttled deterministically, ensuring that admitted traffic continues to meet latency objectives.

Concurrency policies are tuned based on domain-specific characteristics, including request complexity, cache effectiveness, and downstream service limits.

Backpressure and Load Regulation

Explicit backpressure mechanisms allow MCP servers to signal overload conditions to the orchestrator. In response, the platform can defer non-critical requests or apply graceful degradation strategies for lower-priority interactions.

Operational Outcome In production environments, these mechanisms yield stable P95 and P99 latency profiles, predictable throughput scaling, and controlled degradation under stress. The modest increase in median latency is offset by improved tail stability and operational predictability as system complexity grows.

Cost Predictability and Capacity Planning

Cost Structure Decomposition

The MCP platform's cost profile is driven by three primary components: agent inference, MCP server execution infrastructure, and downstream system utilization. Each component scales independently, enabling explicit attribution of cost to individual domains and workflows.

Agent-related costs scale with request volume, reasoning depth, and context size. Because agents remain stateless and are invoked only at orchestration boundaries, inference cost grows proportionally with sustained traffic rather than overall system complexity. MCP server costs scale with domain throughput and are bounded by explicit concurrency limits enforced by the orchestrator.

Domain-Level Capacity Planning

In contrast to monolithic deployments, capacity planning in MCP-based systems occurs at the domain boundary. Search, pricing, inventory, recommendations, and fulfillment are provisioned independently based on observed traffic patterns, latency sensitivity, and business criticality.

This model avoids global over-provisioning driven by localized demand spikes and enables targeted investment where capacity directly impacts user experience and revenue.

Infrastructure Elasticity

Stateless MCP servers enable rapid elasticity. Instances can scale up during traffic surges and scale down without coordination or state migration. Scaling decisions are driven purely by request rates and latency thresholds, improving cost efficiency under volatile retail workloads.

Cost Predictability and Capacity Planning

Cost Structure Decomposition

The MCP platform's cost profile is driven by three primary components: agent inference, MCP server execution infrastructure, and downstream system utilization. Each component scales independently, enabling explicit attribution of cost to individual domains and workflows.

Agent-related costs scale with request volume, reasoning depth, and context size. Because agents remain stateless and are invoked only at orchestration boundaries, inference cost grows proportionally with sustained traffic rather than overall system complexity. MCP server costs scale with domain throughput and are bounded by explicit concurrency limits enforced by the orchestrator.

Domain-Level Capacity Planning

In contrast to monolithic deployments, capacity planning in MCP-based systems occurs at the domain boundary. Search, pricing, inventory, recommendations, and fulfillment are provisioned independently based on observed traffic patterns, latency sensitivity, and business criticality.

This model avoids global over-provisioning driven by localized demand spikes and enables targeted investment where capacity directly impacts user experience and revenue.

Infrastructure Elasticity

Stateless MCP servers enable rapid elasticity. Instances can scale up during traffic surges and scale down without coordination or state migration. Scaling decisions are driven purely by request rates and latency thresholds, improving cost efficiency under volatile retail workloads.

Business Value and Trade-Offs

Cost versus Conversion Impact

Although modular MCP architectures introduce additional infrastructure layers, they provide tighter control over tail latency, which has a measurable impact on user engagement. Across large retail platforms, P95 and P99 latency have a stronger correlation with conversion outcomes than average response time.

By isolating degraded domains and enforcing bounded execution, MCP-based systems protect revenue-critical user journeys such as search discovery and checkout flows, even when individual dependencies experience partial degradation.

Predictable Scaling Economics

Reactive scaling strategies in monolithic systems often result in cost volatility driven by short-lived traffic spikes. MCP-based platforms exhibit more predictable scaling behavior, where cost growth aligns with sustained business expansion rather than transient load fluctuations.

This predictability simplifies budgeting, capacity forecasting, and executive decision-making during high-impact retail events such as seasonal promotions.

Operational Cost Considerations

Separating reasoning from execution reduces operational complexity. Failures are localized to individual domains, recovery procedures are narrowly scoped, and execution paths are explicitly captured. These properties reduce diagnostic effort and ongoing operational overhead without requiring specialized runtime coupling.

Strategic Implication The primary trade-off introduced by MCP architectures is increased structural explicitness in exchange for long-term cost stability, predictable scaling, and controlled tail latency. For enterprise-scale platforms, this trade-off becomes increasingly favorable as traffic volume, domain complexity, and agent usage grow.

Multi-Agent Coordination Model

Role Specialization

The MCP platform adopts a role-specialized agent model rather than a single general-purpose agent. Each agent is scoped to a well-defined domain such as search, recommendations, pricing, or inventory. This specialization reduces reasoning complexity and enables domain-optimized prompting, evaluation logic, and fallback strategies.

A central workflow agent resolves user intent and delegates domain-specific subtasks to specialized agents. This hierarchical coordination pattern prevents combinatorial growth in reasoning paths as system capabilities expand.

Agent Invocation Boundaries

Agents never invoke each other directly. All interactions are mediated through the orchestrator, enforcing explicit execution boundaries and eliminating implicit coupling between agent implementations.

Each agent invocation is treated as a discrete, auditable step with structured inputs and outputs. This allows workflows to be paused, replayed, or partially re-executed without impacting concurrent requests.

Concurrency Management

Concurrency limits are enforced per agent and per MCP server. Limits are tuned based on domain criticality and downstream system capacity, ensuring that high-volume domains do not starve lower-throughput but operationally critical capabilities.

Workflow Scaling and Cost Characteristics

Lightweight MCP Server Execution Model

MCP servers are deployed as lightweight, stateless execution services following the Model Context Protocol runtime model. Each server exposes a bounded set of tools, resources, and prompts through strongly typed MCP contracts, without embedding application-level orchestration or agent reasoning. In production environments, maintaining MCP servers for low-throughput generative AI workloads incurs minimal infrastructure overhead. Typical deployments supporting limited traffic volumes (on the order of tens of requests per second) require modest baseline capacity, with costs remaining small relative to overall generative AI platform spend. As traffic increases, MCP servers scale elastically, ensuring that infrastructure cost grows proportionally with sustained demand rather than idle capacity.

LLM Cost Dominance and Traffic Gating

End-to-end system cost is dominated by large language model inference, which can reach tens or hundreds of thousands of dollars per month at scale. As a result, agentic workflows are intentionally exposed to a controlled percentage of users, targeting high-value or complex interactions rather than full traffic coverage.

Because MCP server traffic scales with gated agent usage rather than raw user volume, infrastructure requirements remain modest. This allows enterprises to adopt MCP incrementally without significant baseline infrastructure expansion.

Scaling Implications

Workflow throughput scales by increasing concurrent workflow instances rather than deepening execution chains. Stateless MCP servers and explicit orchestration ensure predictable scaling behavior even as additional agents, tools, and prompt variants are introduced.

Operational Outcome In practice, this model enables multi-agent workflows to scale safely while keeping infrastructure cost low and predictable. MCP servers introduce minimal overhead relative to LLM inference, making them a practical and economical foundation for enterprise-scale agentic systems.

Failure Modes in MCP Systems

Tool Execution Failures

In MCP-based systems, failures primarily originate from external dependencies, including downstream service unavailability, schema violations, timeout conditions, and partial data responses. These conditions are treated as explicit execution outcomes rather than exceptional control-flow paths.

Each MCP server returns structured error responses that encode failure type, severity, and retry eligibility. Errors are persisted alongside successful results in the execution context, enabling agents to reason about failures without restarting workflows or re-invoking unrelated tools.

Agent Reasoning Failures

Agent-level failures arise from incorrect assumptions, insufficient context, or conflicting objectives during planning. Because agents do not directly perform side-effecting operations, reasoning failures do not corrupt execution state.

Agents may re-plan, request additional context, or invoke alternative tools using the same persisted execution history. This separation avoids tightly coupled failure modes common in monolithic agent implementations where reasoning and execution are interleaved.

Context Integrity Guarantees

All intermediate state is written immutably to the context store. Context entries are versioned and append-only, preventing partial updates or inconsistent writes from propagating across execution steps.

Failed operations never overwrite previously validated context, preserving a consistent execution history suitable for recovery and replay.

Failure Modes in MCP Systems

Tool Execution Failures

In MCP-based systems, failures primarily originate from external dependencies, including downstream service unavailability, schema violations, timeout conditions, and partial data responses. These conditions are treated as explicit execution outcomes rather than exceptional control-flow paths.

Each MCP server returns structured error responses that encode failure type, severity, and retry eligibility. Errors are persisted alongside successful results in the execution context, enabling agents to reason about failures without restarting workflows or re-invoking unrelated tools.

Agent Reasoning Failures

Agent-level failures arise from incorrect assumptions, insufficient context, or conflicting objectives during planning. Because agents do not directly perform side-effecting operations, reasoning failures do not corrupt execution state.

Agents may re-plan, request additional context, or invoke alternative tools using the same persisted execution history. This separation avoids tightly coupled failure modes common in monolithic agent implementations where reasoning and execution are interleaved.

Context Integrity Guarantees

All intermediate state is written immutably to the context store. Context entries are versioned and append-only, preventing partial updates or inconsistent writes from propagating across execution steps.

Failed operations never overwrite previously validated context, preserving a consistent execution history suitable for recovery and replay.

Recovery and Retry Strategies

Policy-Driven Retries

Retries are managed centrally by the orchestrator and governed by explicit policies. Each MCP tool declares retry eligibility, backoff behavior, and maximum attempt limits. Retries are restricted to idempotent operations to prevent duplicate effects.

This approach ensures bounded retry behavior and avoids uncontrolled retry loops under partial failure conditions.

Partial Workflow Replay

Because execution context is materialized between steps, workflows can resume from the last successful checkpoint. Partial replay avoids re-running expensive agent reasoning or upstream tool invocations that have already completed.

This capability simplifies operational recovery during downstream outages, deployments, or transient infrastructure instability.

Graceful Degradation

When certain capabilities are unavailable, agents may degrade behavior by skipping optional steps, returning partial responses, or relying on cached results. Degradation policies are domain-specific and encoded declaratively.

Operational Resilience Together, explicit failure modeling, bounded retries, and partial replay enable MCP-based systems to recover predictably from partial failures. Recovery actions are localized to individual execution steps, preserving stable end-to-end latency characteristics under degraded conditions.

Security Model

Tool-Level Authorization

All MCP tool invocations are mediated by the orchestrator and protected by explicit authorization policies. Agents never hold credentials for downstream systems. Instead, each request is authenticated, authorized, and evaluated against policy before execution.

This design prevents unauthorized access even if agent reasoning logic produces unexpected execution intents. Authorization logic is centralized, versioned, and applied uniformly across all domains.

Least-Privilege Execution

MCP servers are provisioned with narrowly scoped permissions aligned to their declared capabilities. Access is constrained by domain, operation type, and deployment environment.

This approach reduces accidental privilege escalation and limits the impact of misconfiguration or downstream compromise.

Workflow Isolation

Each workflow executes within an isolated context namespace. Context data is never shared implicitly across requests, preventing cross-workflow data leakage, unintended inference, or state contamination.

Governance and Compliance

Auditable Execution Records

All agent decisions, tool invocations, and context mutations are recorded as structured execution records. These records capture inputs, outputs, timestamps, and policy decisions.

Auditing is deterministic and complete, enabling post-incident analysis, debugging, and compliance validation without relying on sampling or heuristic logging.

Policy as Configuration

Execution policies including authorization rules, rate limits, timeout budgets, and retry eligibility are defined declaratively and versioned independently from agent logic. Policy changes can be applied without redeploying agents or MCP servers.

Regulatory Alignment

Centralized enforcement of access control and execution policies simplifies alignment with regulatory requirements related to access accountability, data handling, and operational traceability.

Limitations and Trade-Offs

Architectural Explicitness

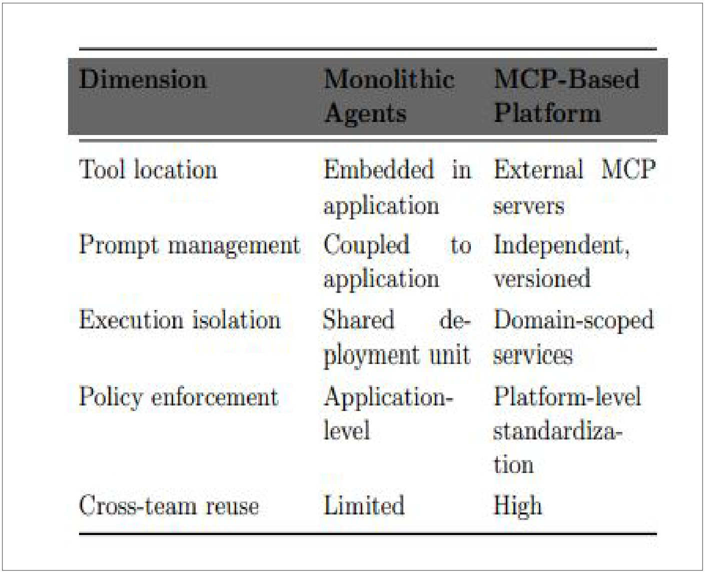
MCP-based platforms introduce additional architectural components compared to monolithic agent implementations. Orchestration layers, context persistence, and domain-specific servers increase system explicitness and operational surface area.

This trade-off is intentional. The platform favors long-term maintainability, policy control, and predictable execution behavior over minimal architectural footprint, which is often insufficient at enterprise scale.

Comparative Analysis of Agent Architectures

Architectural Approaches

Enterprise agent systems have evolved through three dominant architectural patterns: monolithic agent applications, graph-based agent frameworks, and MCP-based platforms. Each approach represents a different trade-off between simplicity, control, and scalability.



Dimension	Monolithic Agents	MCP-Based Platform
Tool location	Embedded in application	External MCP servers
Prompt management	Coupled to application	Independent, versioned
Execution isolation	Shared deployment unit	Domain-scoped services
Policy enforcement	Application-level	Platform-level standardization
Cross-team reuse	Limited	High

Monolithic designs tightly couple prompts, tools, and business logic, making versioning and reuse difficult. MCP platforms externalize these concerns, enabling independent evolution and shared consumption across applications.

Comparison with Graph-Based Agent Frameworks

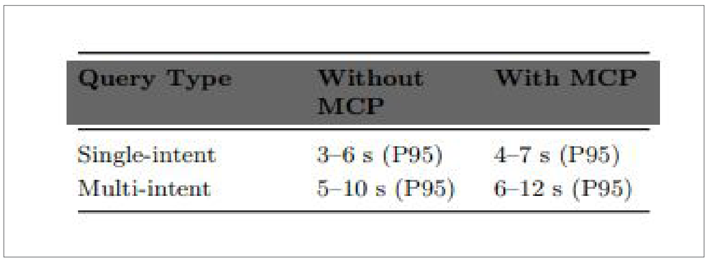
Graph-based frameworks (e.g., DAG or state-machine driven agents) improve structure but still embed execution logic within the agent runtime. MCP differs by treating execution as a service boundary rather than a graph edge.

This distinction allows MCP platforms to enforce operational policies and failure containment without constraining agent reasoning expressiveness.

SLatency, Cost, and Operational Trade-Offs

Latency Characteristics

MCP-based execution introduces additional latency due to orchestration, context persistence, and network boundaries. However, this overhead is bounded and predictable.



Query Type	Without MCP	With MCP
Single-intent	3–6 s (P95)	4–7 s (P95)
Multi-intent	5–10 s (P95)	6–12 s (P95)

The observed increase of 1–2 seconds is primarily attributable to explicit context handling, agent deliberation depth, and MCP tool invocation overhead, rather than LLM inference alone.

Cost Considerations

MCP servers introduce a predictable and bounded infrastructure overhead relative to monolithic agent deployments. In current production use cases, sustained request rates typically fall in the 10–30 requests per second range per domain, driven by product scope and budget allocation rather than architectural constraints.

MCP servers are lightweight and horizontally scalable, and can support higher throughput as demand grows. In practice, overall system cost is dominated by LLM inference, so MCP-based execution is applied selectively to high-value user flows and scaled incrementally as business requirements evolve.

By contrast, LLM inference dominates overall spend by orders of magnitude. Consequently, MCP architectures are economically viable when applied to selective, high-value traffic segments rather than full population coverage.

Trade-Off Summary

MCP platforms trade minimal added latency and infrastructure complexity for significant gains in prompt governance, tool reuse, deterministic execution, and enterprise operability. For large-scale agentic systems, this trade-off favors MCP-based designs as system scope and organizational complexity grow.

Observations from Production Deployment

Operational Stability

Production telemetry indicates that execution latency in MCP-based systems is primarily influenced by agent reasoning depth, the underlying language model, the number of tools invoked, and the size of input and output context. As workflows involve additional domains or tools whether executed sequentially or in parallel end-to-end latency increases in a predictable manner.

Isolating execution into MCP servers does not eliminate this latency growth, but it ensures that performance characteristics remain stable and explainable. Latency increases are attributable to explicit factors such as model selection, token budgets, orchestration strategy, and tool invocation patterns, rather than hidden coupling or uncontrolled side effects within the application. Under normal operating conditions, agent workflows execute within expected latency envelopes given their reasoning complexity and tool usage, allowing teams to reason about performance trade-offs explicitly as workflows evolve.

This structural isolation provides a clear operational boundary: execution behavior in one domain is governed independently from others. While the evaluation period did not observe systemic performance regressions, the architecture ensures that any future domain-specific slowdowns or maintenance events would remain contained without impacting unrelated workflows.

Developer Velocity

Teams evolve agent reasoning logic independently from execution services. Updates to prompts, intent classification, routing heuristics, or decision policies do not require coordinated redeployment of MCP servers.

This decoupling reduces release coordination overhead and enables faster iteration on agent behavior while maintaining stable execution interfaces and operational safeguards.

Cost-to-Performance Ratio

Introducing MCP servers adds a modest increase in infrastructure cost relative to monolithic agent deployments. This increase is primarily attributable to the deployment and operation of dedicated execution services per domain. In practice, this additional cost remains well within established infrastructure budgets.

The benefit of this trade-off is improved cost transparency and control. Execution costs scale primarily with tool invocation volume and domain-level traffic rather than prompt size or agent reasoning complexity alone. This separation enables predictable capacity planning and avoids unexpected cost amplification as agent logic evolves.

Overall, the MCP-based model prioritizes controlled cost growth and operational clarity over absolute infrastructure minimization, aligning well with enterprise-scale budgeting and governance requirements.

Summary Empirical observations confirm that MCP-based architectures provide execution isolation, and favorable cost–performance trade-offs under sustained production load.

Latency Characteristics Under Load

The following measurements were collected under controlled production-like conditions. Each request includes full conversational context, agent reasoning, tool selection, MCP server execution, and response synthesis.

Parameter	Configuration	Observed Range
Request rate	Sustained traffic	10 TPS
LLM context size	Tokens per request	∼10,000 tokens
MCP server tools	Active tools	8 tools
Agent execution	Planning and routing	Included in latency

End-to-End Response Time

Latency measurements reflect full round-trip execution, including agent reasoning, MCP server invocation, downstream tool execution, and response generation.

Query Type	P95 Latency	Notes
Single-intent queries	3–8 seconds	Search, Recs, or Fulfillment
Multi-intent queries	6–12 seconds	Search, Recs, Fulfillment, Project
Context-heavy requests	4–10 seconds	Includes full conversation history

Interpretation End-to-end latency is primarily influenced by agent reasoning depth, context size, and the number of MCP tool invocations rather than raw LLM inference time alone. For multi-intent queries, MCP tools may be executed sequentially or in parallel depending on dependency structure and orchestration policy.

Additional latency arises from context accumulation and coordination overhead, not from MCP server execution itself. Despite this, observed P95 latencies remain stable and bounded, supporting interactive user experiences under realistic enterprise workloads.

Conclusion

This paper presented an MCP-based platform architecture for building enterprise-scale, agent-driven generative AI systems with strong guarantees around latency, correctness, and operational safety. By externalizing tools, prompts, and execution logic into domain-specific MCP servers, the platform replaces monolithic agent implementations with a modular, governed execution model. Agents remain focused on reasoning and planning, while MCP servers provide versioned, reusable access to tools, resources, and prompts across multiple applications and workflows. Explicit context materialization enables deterministic replay, partial workflow recovery, and auditable execution traces without coupling agent logic to execution timing or infrastructure state. Tail-latency governance at P95 and P99 is achieved through bounded orchestration, domain isolation, and controlled tool invocation rather than reliance on average-case optimization.

Taken together, these design choices demonstrate that agentic systems can meet production reliability and governance requirements while preserving flexibility and rapid iteration velocity.

Key Contributions

• An MCP-based execution model that decouples agent reasoning from tools, resources, and prompts

• Externalized, versioned tool and prompt management via domain-specific MCP servers

• Deterministic context persistence enabling replay, recovery, and auditability

• Predictable P95/P99 latency governance under multi-agent, multi-tool workloads

• A cost-aware scaling model aligned with targeted, low-percentage traffic deployment

Final Remark MCP establishes a practical systems abstraction for scaling agentic generative AI beyond experimental prototypes. By treating tools, prompts, and execution as first-class, governable services, the platform bridges the gap between rapid agent innovation and enterprise-grade reliability, cost control, and compliance.

References

1. OpenAI. Model Context Protocol (MCP). 2024.
2. Google. Agentic Systems on Cloud Infrastructure. 2023.
3. Microsoft. Building Reliable and Governed AI Agents. 2023.
4. AWS. Operational Excellence for Generative AI Workloads. 2024.
5. Zaharia et al. Lakehouse Architectures for AI Systems. VLDB, 2021.
6. Chen et al. Managing Tail Latency in Distributed Systems. SOSP, 2021.
7. Meta AI. Toolformer: Language Models That Can Use Tools. NeurIPS, 2022.
8. Google Research. Production Considerations for Large Language Models. 2023