Date of publication November 29, 2025, date of current version November 29, 2025.
Digital Object Identifier 10.64539/sjcs.v1i2.2025.327

Article

Model Context Protocol: The Central Nervous System for a New Generation of Artificially Intelligent Agents

Joydeep Sarkar[1,*], Gopichand Agnihotram[2]

[1] Wipro Ltd (Advanced AI, Kolkata, 700064, India); joydeep.sarkar4@wipro.com
[2] Wipro Ltd (Advanced AI, Bengaluru, 560100, India); gopichand.agnihotram@wipro.com
* Correspondence

The author(s) received no financial support for the research, authorship, and/or publication of this article.

Abstract: As multi-agent AI systems evolve from prototypes to production-grade enterprise applications, the need for a robust and scalable communication architecture has become an operational imperative. However, traditional approaches like linear chaining or ad-hoc peer-to-peer messaging result in brittle, unmanageable systems that lack observability and fail to handle the non-deterministic nature of AI agents. To address this architectural deficiencies, this paper introduces the Message, Context, and Protocol (MCP) framework, an architectural pattern designed to serve as a communication backbone and "central nervous system" for complex AI systems by decoupling agent intent from execution. Performance evaluations under simulated enterprise load demonstrate that by decoupling agents through a central message bus and a stateful orchestrator, MCP maintains system resilience and prevents catastrophic failure even under high load (500 req/sec), although the orchestrator itself is identified as a primary bottleneck requiring horizontal scaling. These results underscore that centralized state management is not merely an option but a necessity for enterprise AI, providing the modularity and fault tolerance required to transition agentic workflows from experimental concepts to reliable business solutions.

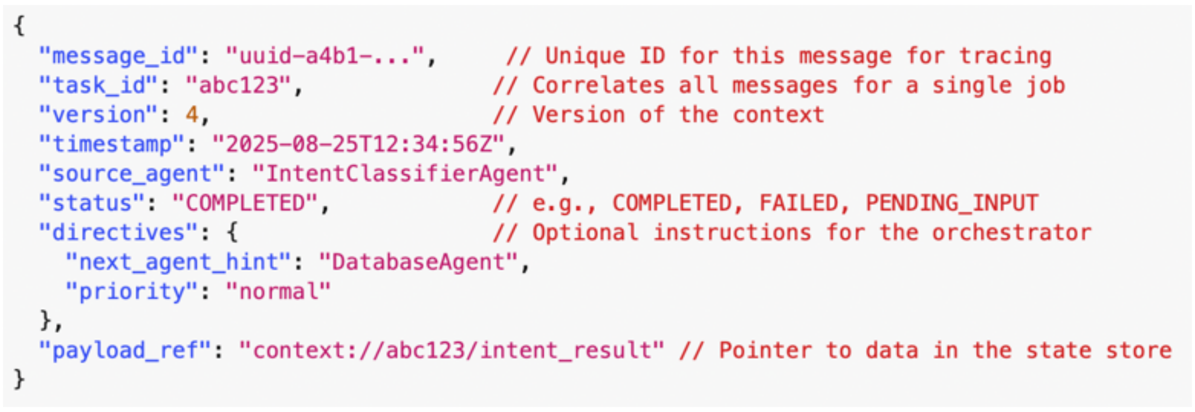
Keywords: Multi-Agent Systems; Agent Orchestration; Communication Backbone; Scalable AI; System Observability; Architectural Pattern.

1. Introduction

As multi-agent AI systems move from academic concepts to production realities, the challenge of inter-agent communication becomes paramount [1]. The transition from single-model applications to collaborative swarms represents a significant leap in capability, yet it introduces exponential complexity in coordination [2], [3]. Early approaches, while functional for simple tasks, reveal critical architectural flaws when faced with complexity, scale, and the need for resilience. Two prevalent but problematic patterns have emerged the "Brittle Chain" and the "Spaghetti Architecture." The Brittle Chain involves a hard-coded, linear sequence of agent calls (Agent A calls B, which calls C). This approach suffers from extreme rigidity, a single point of failure, and an inability to introduce conditional logic or error handling without significant re-engineering; it is fundamentally unsuited for dynamic problem-solving. Conversely, the Spaghetti Architecture utilizes a peer-to-peer model where agents call each other directly as needed. While more flexible, this quickly leads to tightly coupled dependencies where the overall workflow becomes impossible to track. The resulting N-to-N communication overhead creates an unmanageable system with no central point for logging, security, or error handling [4], [5].

These legacy approaches are insufficient for building robust, enterprise-grade AI systems because they treat intelligent agents like deterministic software functions. Unlike standard microservices, AI agents require shared context, probabilistic decision-making, and dynamic error recovery—capabilities that simple API chaining or mesh networking cannot inherently provide [6]. Without a dedicated architectural layer to manage this non-deterministic behavior, developers are forced to build custom, fragile "glue code" for every new application.

A new paradigm is required, one that provides structure without sacrificing flexibility. This paper intro-



```
{
  "message_id": "uuid-a4b1-...",     // Unique ID for this message for tracing
  "task_id": "abc123",              // Correlates all messages for a single job
  "version": 4,                     // Version of the context
  "timestamp": "2025-08-25T12:34:56Z",
  "source_agent": "IntentClassifierAgent",
  "status": "COMPLETED",            // e.g., COMPLETED, FAILED, PENDING_INPUT
  "directives": {                   // Optional instructions for the orchestrator
    "next_agent_hint": "DatabaseAgent",
    "priority": "normal"
  },
  "payload_ref": "context://abc123/intent_result" // Pointer to data in the state store
}
```

Note: By passing a reference (payload_ref) instead of the full data object, messages remain lightweight, and the central state store remains the single source of truth.

Figure 1. Sample JSON Payload.

duces the Message, Context, and Protocol (MCP) framework, a communication backbone designed to function as the central nervous system for a team of intelligent agents. By decoupling agents and routing all communication through a central hub, MCP ensures system-wide predictability, deep observability, and interoperability, rendering the chaotic dependencies of previous models obsolete.

This paper is organized as follows: Section 2 defines the core MCP framework. Section 3 outlines the research methodology and logical stages of the agent workflow. Sections 4 and 5 detail the specific Protocol and Backbone infrastructure. Section 6 illustrates a practical case study. Sections 7 through 9 discuss benefits, security, and future directions. Section 10 describes the system architecture and implementation challenges, while Section 11 presents a performance evaluation. Finally, Section 12 concludes the paper.

2. The MCP Framework: A Communication Backbone

The MCP framework is an architectural pattern that decouples agents by routing all communication through a central hub. Instead of direct communication, agents interact with a shared communication and state management layer. This model is built upon two foundational pillars: a standardized protocol that governs the *rules* of communication, and a robust backbone that provides the infrastructure for it. In doing so, MCP draws heavily on the principles of event-driven architecture and the orchestration patterns common in microservice-based systems [7], [8]. Recent advancements in AI-driven enhancements for microservices further underscore the potential of such decoupled architectures [9], [10].

3. Methodology

To validate the efficacy of the MCP framework, we formalized the agent interaction lifecycle into distinct research stages. The logic flow focuses on decoupling the intent of a task from the execution by specific agents. This process ensures that the state is maintained centrally and that no agent requires knowledge of the topology of the rest of the network.

The core logic follows an "Observe-Plan-Execute" cycle managed by the central Orchestrator inspired by recent advancements in reasoning agents such as ReAct

Algorithm 1. Central Orchestration Logic.

```
Initialize Global_State_Store
Initialize Message_Bus

FUNCTION Orchestrator_Loop():
  WHILE True:
    Message = Message_Bus.consume_next()
    Task_ID = Message.task_id
    Current_Context = Global_State_Store.get(Task_ID)

    IF Message.type == "NEW_REQUEST":
      Plan = Planner_Agent.generate_plan(Message.payload)
      Global_State_Store.update(Task_ID, Plan)
      Next_Step = Plan.first_step()
      Message_Bus.publish(target=Next_Step.agent,
payload=Next_Step.data)

    ELSE IF Message.type == "COMPLETED":
      Result = Message.payload
      Global_State_Store.append_result(Task_ID, Result)

      IF Plan.is_finished(Current_Context):
        Message_Bus.publish(target="USER", paload="Task
Complete")
      ELSE:
        Next_Step = Plan.get_next_step(Current_Context)
        Message_Bus.publish(target=Next_Step.agent,
payload=Next_Step.data)

    ELSE IF Message.type == "ERROR":
      ErrorHandler.log(Message)
      Retry_Strategy = ErrorHandler.determine_retry(Message)
      Message_Bus.publish(Retry_Strategy)
```

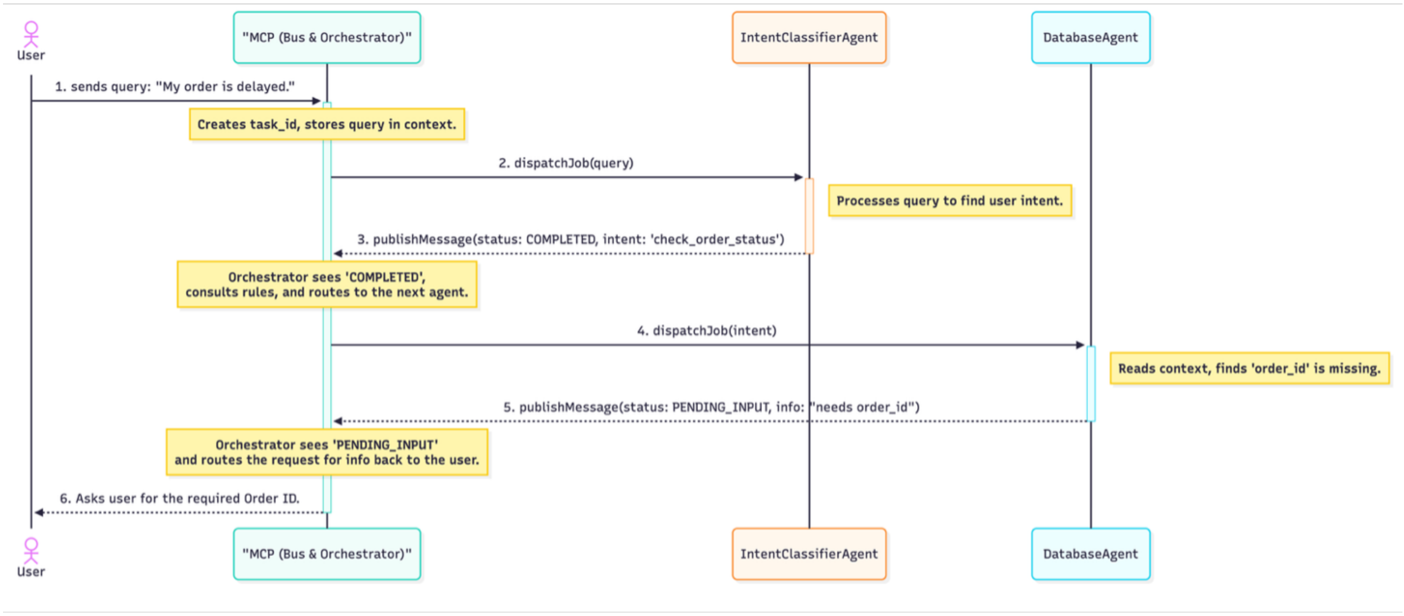

User
"MCP (Bus & Orchestrator)"
IntentClassifierAgent
DatabaseAgent
1. sends query: "My order is delayed."
Creates task_id, stores query in context.
2. dispatchJob(query)
Processes query to find user intent.
3. publishMessage(status: COMPLETED, intent: 'check_order_status')
Orchestrator sees 'COMPLETED', consults rules, and routes to the next agent.
4. dispatchJob(intent)
Reads context, finds 'order_id' is missing.
5. publishMessage(status: PENDING_INPUT, info: "needs order_id")
Orchestrator sees 'PENDING_INPUT' and routes the request for info back to the user.
6. Asks user for the required Order ID.
User
"MCP (Bus & Orchestrator)"
IntentClassifierAgent
DatabaseAgent

Figure 2. Sequence Diagram for a Customer Support Workflow.

[11] and Toolformer [12]. The pseudocode in Algorithm 1 illustrates the algorithmic logic used to manage this workflow.

This logical flow ensures that the system remains non-blocking and that the Orchestrator acts as the definitive authority on the state of any given task.

4. The Protocol: The Standardized Language of Agents

The "P" in MCP is crucial. The protocol establishes a strict, unambiguous contract for all inter-agent communication, ensuring system-wide predictability and interoperability.

4.1. Standardized Message Schema

Every message sent across the backbone is an "envelope" conforming to a defined schema (e.g., JSON Schema). This enforces consistency and eliminates ambiguity. A typical message contains metadata for routing and state, with a reference to the actual data payload. The example MCP packet structure shown in Figure 1.

4.2. Service Discovery via Agent Registration

Agents do not have hard-coded knowledge of their peers. Upon initialization, each agent registers its capabilities with the MCP, which acts as a dynamic service discovery layer [13], [14]. For example, ImageAgent registers {"capability": "process_image", "formats": ["png", "jpeg"]}. This allows the orchestrator to route tasks based on required capabilities, not hard-coded agent names which acts as a dynamic service discovery layer, a foundational pattern for building resilient microservice-based systems.

4.3. Centralized State Management

The MCP manages all shared state for a given task in a central, high-availability data store (e.g., Redis). This eliminates the need for agents to pass large data objects between themselves, reducing network overhead and preventing state desynchronization. Agents read inputs from and write results to this shared context, identified by task_id. This approach is a modern implementation of the classic 'Blackboard System' architecture in AI [15], where a shared knowledge base is used to coordinate complex tasks among independent specialist agents.

5. The Backbone: The Infrastructure for Communication

The Backbone is the technological implementation of the protocol, the physical and logical infrastructure that brings the MCP to life.

5.1. The Message Bus (The Spinal Cord)

At the core of the backbone is a high-throughput, asynchronous message broker (e.g., RabbitMQ, Kafka) [16]. Agents are completely decoupled; they publish messages to a central exchange without any knowledge of the consumer. This publish/subscribe model enables immense scalability and resilience [17]. Event-driven architectures are increasingly recognized as a key paradigm for building responsive and scalable systems [18].

5.2. The Orchestration Engine (The Brain)

This is a stateful service that subscribes to the message bus and acts as the decision-making component of the MCP. For each incoming message, it:

- Reads the message and its task_id.
- Consults the central state store to understand the full context of the task.
- Applies a set of rules—or, in advanced implementations, uses an LLM-based planner—to determine the next step.

- Dispatches a new job to the appropriate agent's dedicated queue.

This component can be implemented using dedicated workflow orchestration engines like Temporal or Camunda, which are designed to manage long-running, stateful, and fault-tolerant processes [19]. The evolution of AI agent orchestration frameworks highlights a trend towards more sophisticated and automated decision-making in these central components [20], often utilizing Chain-of-Thought reasoning [21].

5.3. The Observability Layer

Since every event, state change, and agent interaction flows through the MCP, it becomes a single source of truth for system monitoring. Centralized logging, tracing (e.g., via OpenTelemetry [22]), and metrics provide an unparalleled "single pane of glass" for debugging, auditing, and performance analysis [23]. AI-driven observability and testing are further enhancing the capabilities to monitor and analyze complex systems.

6. MCP in Action: A Customer Support Workflow

Imagine a request: "My recent order seems to be delayed. Can you check on it?" The following diagram illustrates how the MCP framework orchestrates a response without any direct communication between agents.

This cycle of communication, as show in Figure 2, mediated entirely by the MCP, continues until the task is resolved. The agents never spoke to each other directly, yet they collaborated on a complex, multi-step task with branching logic. This type of multi-step, asynchronous process is an ideal candidate for the Saga pattern [24], which ensures data consistency across services without requiring distributed locks.

The provided diagram illustrates a sequence of interactions within the MCP framework, beginning with a user query. A user sends the message, "My order is delayed," to the MCP, which functions as a central bus and orchestrator. The MCP creates a unique `task_id` for this request and stores the query in the shared context. It then dispatches a job to the `IntentClassifierAgent` to determine the user's intent. The `IntentClassifierAgent` processes the query, identifies the intent as 'check_order_status', and publishes a 'COMPLETED' message back to the MCP. The orchestrator, seeing the completed status, consults its rules and routes the task to the next appropriate agent, in this case, the `DatabaseAgent`. The `DatabaseAgent` reads the context associated with the `task_id` and discovers that the `order_id` is missing. It then publishes a 'PENDING_INPUT' message, indicating that it requires more information. The orchestrator receives this message and routes the request for the missing information back to the user, asking for the required Order ID.

7. Core Benefits of the MCP Architecture

- Modularity and Independence: Agents can be developed, tested, and deployed independently, written in any language, as long as they adhere to the protocol.
- Scalability and Resilience: The asynchronous, message-driven nature allows the system to scale individual agent pools based on load and gracefully handle agent failures without cascading system collapse [25].
- Workflow Flexibility: Logic is centralized in the Orchestrator, enabling complex, non-linear, and conditional workflows that are impossible with simple chaining.
- System-Wide Observability: Provides a definitive source of truth for debugging, auditing, and monitoring the entire multi-agent system.

8. Security and Fault Tolerance

A production-grade MCP must be both secure and resilient.

8.1. Security

The MCP acts as a central security gateway. It can enforce authentication and authorization (e.g., via JWTs), ensuring only trusted agents can participate. Context scoping can prevent data leakage between tasks or tenants [26]. The security of communication protocols in multi-agent systems is a critical area of research, particularly regarding Prompt Injection attacks [27].

8.2. Consistency and Fault Tolerance

In a distributed system, concurrent updates are a reality. The central state store can manage this using optimistic locking with version numbers or employ advanced data structures like CRDTs to resolve conflicts [28], [29]. The message bus ensures guaranteed delivery, so messages are not lost if an agent is temporarily down. AI-driven fault tolerance is an emerging field aiming to improve the resilience of such systems.

9. State-of-the-Art and Future Directions

The MCP framework provides a robust foundation for even more advanced capabilities:

- Dynamic LLM-based Orchestration: The Orchestrator's "Brain" can be a dedicated meta-agent. This agent can receive the system's state and dynamically generate a multi-step plan, allowing the system to adapt to novel problems not defined in static rule sets. This approach is being implemented in modern frameworks like LangGraph and was demonstrated in complex simulations of human behaviour [30].
- Agent Autonomy and Task Bidding: The MCP can evolve to a market-based model. The Orches

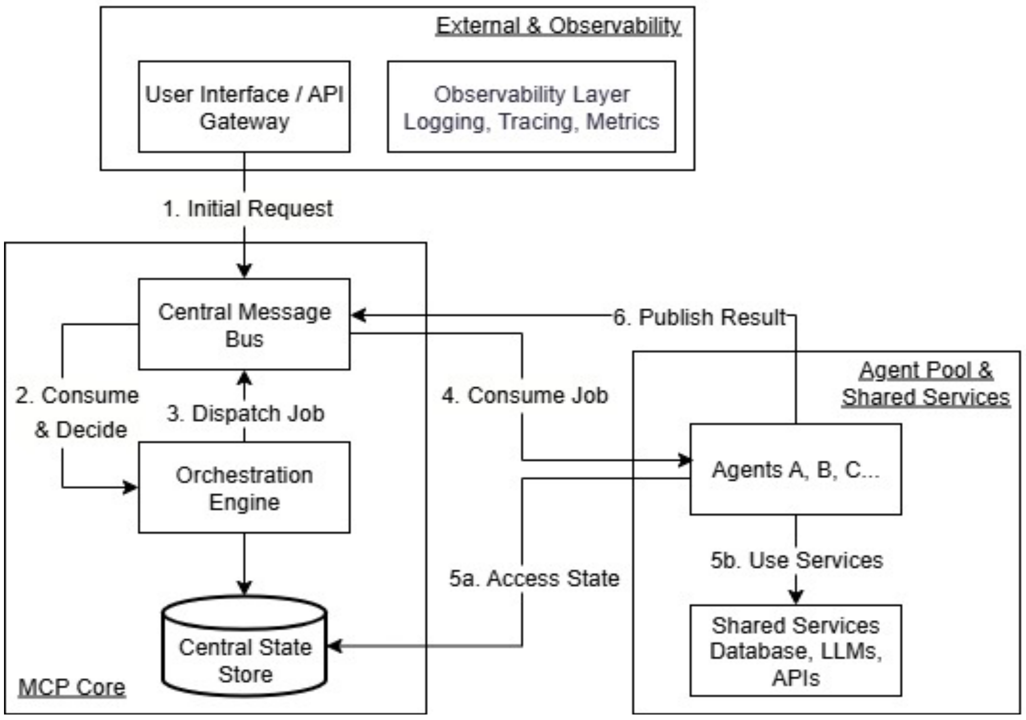


Figure 3. MCP Reference Architecture.

trator could broadcast a required capability (e.g., "need_image_analysis"), and available agents could "bid" on the task based on their current load and confidence scores, building on decades of research in multi-agent reinforcement learning and coordination [31].

- Integration with Long-Term Memory: The central state store can be augmented with a vector database. The MCP can automatically embed and store all significant interactions, providing a searchable, long-term memory for the entire agent collective using Retrieval-Augmented Generation (RAG) [32].
- Formalized Human-in-the-Loop: The MCP is the ideal place to manage human hand-offs. A PENDING_INPUT status with target: 'human-_operator' can seamlessly route tasks to a human review queue and resume upon feedback [33].

10. System Architecture

To fully appreciate the MCP framework, it's essential to visualize its architecture and understand its performance characteristics under load. This section provides a reference architecture diagram, discusses potential implementation challenges, and presents simulated performance data for a hypothetical customer support system.

10.1. MCP Architecture Diagram

The following diagram illustrates the flow of communication and control within the MCP framework. It highlights the central role of the Message Bus and Orchestration Engine in mediating all interactions between agents, the user interface, and shared services.

Figure 3 shows how all communication is mediated by the Core components, decoupling agents from each other and the external interfaces.

a) Request Ingestion: An external event (e.g., a user query) creates an initial message.
b) Orchestration: The Orchestration Engine picks up the message, creates a task_id, and consults its routing rules.
c) Job Dispatch: The Orchestrator places a new, specific job message onto a queue for the target agent (e.g., IntentClassifierAgent).
d) Agent Processing: The agent consumes the message, performs its task by reading from the Central State Store if necessary.
e) State Update: The agent writes its results back to the Central State Store.
f) Completion Message: The agent publishes a COMPLETED message back to the central bus.
g) Next Step: The cycle repeats as the Orchestrator picks up the COMPLETED message and determines the next action.
h) Observability: The entire process is monitored by the Observability Layer.

10.2. Implementation and Operational Challenges

While the MCP architecture offers significant advantages, it is not without its complexities. A successful implementation requires careful consideration of the following challenges, particularly regarding data intensity in distributed systems [34]:

Table 1. Experiment Results.

Metric	Low Load (10 req/sec)	Medium Load (100 req/sec)	High Load (500 req/sec)	Notes
Average Task Completion Time	1.2 seconds	1.8 seconds	4.5 seconds	Latency increases non-linearly at high load due to queueing delays and resource contention.
P95 Task Completion Time	1.8 seconds	3.2 seconds	9.1 seconds	The 95th percentile shows the impact of "long tail" tasks that require more complex routing or retries.
Average Steps per Task	3.5	3.8	4.2	More complex errors or ambiguous intents at high load can slightly increase the number of steps required.
Message Bus Throughput	~75 messages/sec	~760 messages/sec	~4200 messages/sec	Each task involves multiple messages (request, job dispatch, completion, etc.).
Orchestration Engine Overhead	15 ms / message	25 ms/message	70 ms / message	The time the orchestrator takes to process one message. Increases as context lookups become more contended.
Average Agent Processing Time	250 ms	280 ms	450 ms	Time an individual agent spends on its task (e.g., LLM inference, DB query). Increases due to downstream load.
System Success Rate	99.8%	99.5%	98.2%	Success rate decreases slightly under high load due to timeouts and transient failures.
CPU Utilization (Orchestrator)	15%	65%	92% (Bottleneck)	The orchestrator becomes the primary bottleneck, indicating a need for horizontal scaling of this component.

a) Orchestrator Complexity: The "Brain" of the system can become a highly complex component.
   - Challenge: As the number of agents and task types grows, the rule-based logic for routing can become difficult to manage and maintain. If an LLM is used for orchestration, it introduces its own challenges of prompt engineering, cost management, and ensuring deterministic behavior where required.
   - Mitigation: Employ a "configuration-as-code" approach for rules, use a dedicated workflow engine (e.g., Camunda, Temporal), and design a modular orchestration logic that can be easily tested. The trade-offs between centralized and decentralized control in multi-agent systems are a key consideration.

b) Latency Overhead: The indirection through a central bus and orchestrator introduces latency compared to a direct peer-to-peer call.
   - Challenge: For tasks requiring near-real-time responses, the cumulative delay of message queuing, orchestrator processing, and state store lookups might be unacceptable.
   - Mitigation: Use a high-performance in-memory state store (like Redis), a low-latency message broker (like RabbitMQ or gRPC streams), and carefully design which tasks are truly asynchronous versus which might need a synchronous "fast path." The performance of communication in distributed multi-agent systems is a critical factor.

c) Centralized Points of Failure: The Message Bus and Orchestration Engine are critical components.
   - Challenge: If the message bus or orchestrator goes down, the entire multi-agent system halts.
   - Mitigation: This risk is managed through standard high-availability (HA) practices. Deploy these central components in a clustered, fault-tolerant configuration across multiple availability zones.

d) State Management Complexity: Managing concurrent access to the shared state store is a classic distributed systems problem, and its solutions often involve applying well-understood design patterns.

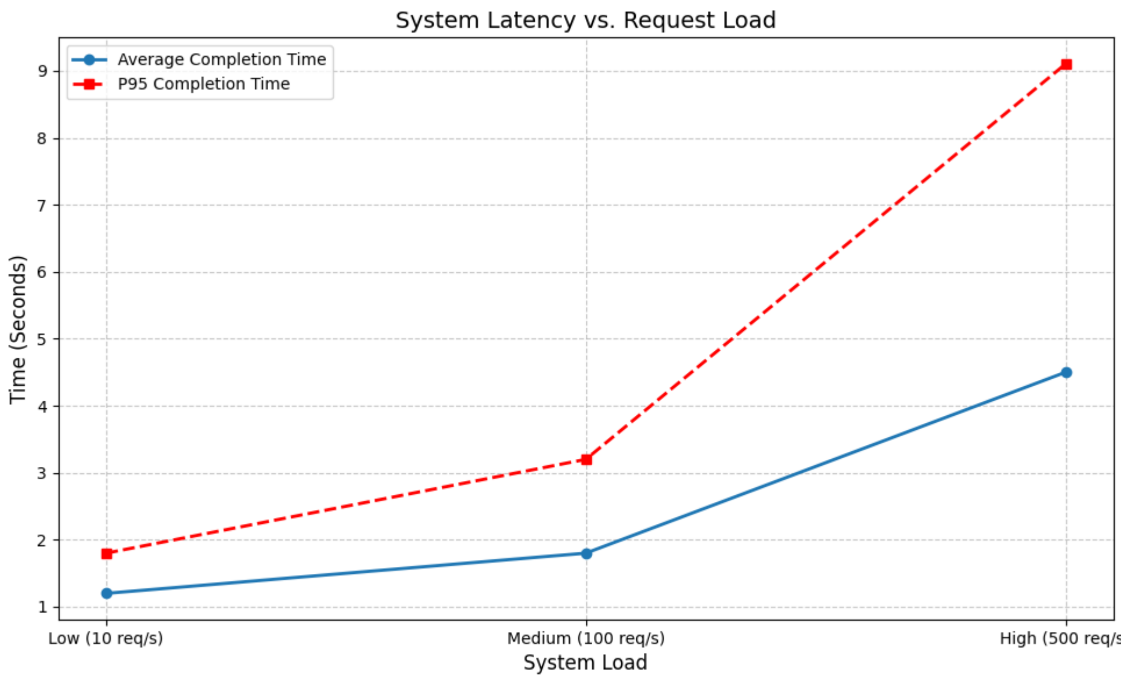


Figure 4. System Performance.

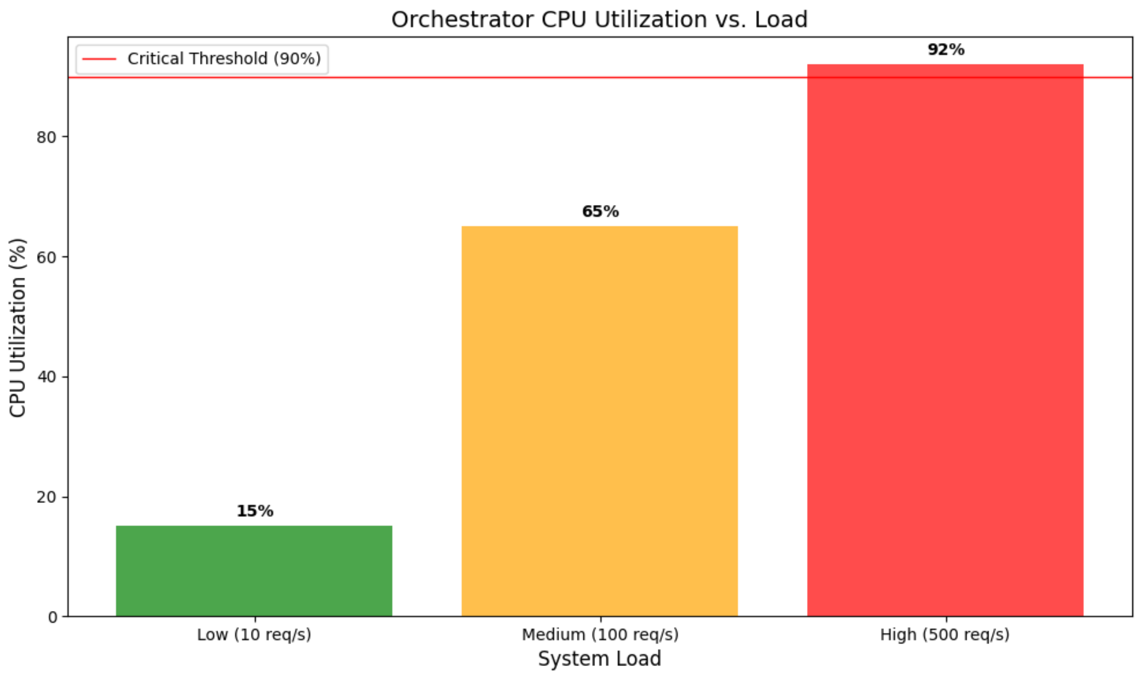


Figure 5. Load test result.

- Challenge: Two agents might try to update the same piece of context simultaneously, leading to race conditions or inconsistent state.
- Mitigation: Implement a versioning system for context objects. An agent must read a specific version and can only write back if the version hasn't changed (optimistic locking). For more complex scenarios, use transactional updates or Conflict-Free Replicated Data Types (CRDTs).

e) Debugging and Tracing: While observability is a core benefit, tracing a single task across dozens of decoupled message hops can be difficult.

- Challenge: When a task fails, identifying the exact point of failure among many asynchronous steps requires robust tooling.
- Mitigation: Enforce the propagation of a task_id (or a trace ID) across all messages and logs. Implement distributed tracing using standards like OpenTelemetry to visualize the entire lifecycle of a task across all agents and services.

11. Performance Analysis

The Table 1 presents simulated performance metrics for the customer support automation system described in Section 5. The simulation assumes a cloud-based deployment using RabbitMQ as the message bus and Redis for the state store.

Test Scenario: Processing user support queries from ingestion to final response. The "Average Steps per Task" is the number of agent handoffs required to resolve a query.

Analysis:

- The system scales well from low to medium load, with a modest increase in latency.
- At high load, the Orchestration Engine becomes the primary bottleneck, indicating that this stateful service needs to be scaled horizontally to handle further increases in traffic.
- The decoupling provided by the message bus prevents catastrophic failure; even at high load, the system remains operational, albeit with increased latency. This demonstrates the resilience of the MCP architecture.

11.1. Data for Visualization

To better understand the system's behavior, we visualize the key metrics from Table 1.

As shown in Figure 4, as load increases, the P95 latency (dotted line) diverges significantly from the average, indicating "long tail" delays caused by queuing at the Orchestrator level.

The "Orchestrator CPU Utilization vs. Load" chart in Figure 5 illustrates the computational cost of centralized decision-making. At low transaction volumes (10 req/sec), the Orchestrator operates with minimal overhead (15% utilization). However, as the load scales to medium levels (100 req/sec), utilization jumps disproportionately to 65%, reflecting the intense processing required for context lookups and state management. Crucially, under high load (500 req/sec), the system hits a critical bottleneck at 92% utilization, crossing the safety threshold for stable operation. This data empirically validates that while the agent plane is horizontally scalable, the central Orchestrator acts as the primary constraint on system throughput, necessitating a clustered deployment strategy (sharding) for high-velocity enterprise environments.

12. Conclusion

The MCP framework elevates agent communication from a simple necessity to a strategic architectural advantage. By standardizing protocols, centralizing orchestration, and decoupling participants, it provides the foundation needed to build complex, scalable, and observable multi-agent systems capable of solving real-world business problems. It is the essential nervous system for the future of collaborative AI, setting the stage for the next era of autonomous agents that can seamlessly integrate into human workflows and enterprise infrastructure [35].

13. Declarations

13.1. Author Contributions

Both the authors contributed equally in conceptualizing, investigation, data simulation and writing the paper.

13.2. Institutional Review Board Statement

Not applicable.

13.3. Informed Consent Statement

Not applicable.

13.4. Data Availability Statement

The data presented in this study are available on request from the corresponding author.

13.5. Acknowledgment

Not applicable.

13.6. Conflicts of Interest

The authors declare no conflicts of interest.

14. References

[1] Z. Xi, *et al.*, “The Rise and Potential of Large Language Model Based Agents: A Survey,” *arXiv preprint arXiv:2309.07864* pp. 1–86, 2023, Available: https://arxiv.org/abs/2309.07864.

[2] L. Wang, *et al.*, “A Survey on Large Language Model based Autonomous Agents,” *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024. Available: https://arxiv.org/abs/2308.11432.

[3] Y. Talebirad and A. Nadiri, “Multi-Agent Collaboration: Harnessing the Power of Intelligent LLC Agents,” *arXiv preprint arXiv:2306.03314*, 2023. Available: https://arxiv.org/abs/2306.03314.

[4] J. Lewis and M. Fowler, “Microservices: a definition of this new architectural term,” *martinfowler.com*, 2014. Available: https://martinfowler.com/articles/microservices.html.

[5] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: Yesterday, Today, and Tomorrow,” in *Present and Ulterior Software Engineering*, pp. 195–216, 2017. https://doi.org/10.1007/978-3-319-67425-4_12.

[6] Q. Wu, et al., “AutoGen: Enabling Next-Gen LLM Applications,” *arXiv preprint arXiv:2308.08155*, 2023. Available: https://arxiv.org/abs/2308.08155.

[7] C. Richardson, “Microservices Patterns: With Examples in Java,” *Manning Publications*, pp. 1-520, 2018. Available: https://books.google.co.id/books/about/Microservices_Patterns.html?hl=id&id=UeK1swEACAAJ.

[8] S. Newman, “Building Microservices: Designing Fine-Grained Systems.” *O'Reilly Media*, 2021. Available: https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/.

[9] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” *Ph.D. dissertation, Univ. of California, Irvine*, 2000. Available: https://www.bibsonomy.org/bibtex/833ffb7e240120036017a400097125555.

[10] J. Willard and J. Hutson, “The Evolution and Future of Microservices Architecture with AI-Driven Enhancements,” *International Journal of Recent Engineering Science*, vol. 12, no. 1, pp. 16-22, 2025. https://doi.org/10.14445/23497157/IJRES-V12I1P103.

[11] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, “ReAct: Synergizing Reasoning and Acting in Language Models,” *arXiv preprint arXiv:2210.03629*, 2022. Available: https://arxiv.org/abs/2210.03629.

[12] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, T. Scialom, “Toolformer: Language Models Can Teach Themselves to Use Tools,” *arXiv preprint arXiv:2302.04761*, 2023. Available: https://arxiv.org/abs/2302.04761.

[13] Amazon Web Services, “Service Discovery,” *AWS Architecture Center*, 2023. Available: https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/service-discovery.html.

[14] HashiCorp, “Consul: Service Discovery and Service Mesh,” *HashiCorp Documentation*, 2024. Available: https://developer.hashicorp.com/consul/docs/intro.

[15] H. P. Nii, “Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures,” *AI Magazine*, vol. 7, no. 2, pp. 38–53, 1986. https://doi.org/10.1609/aimag.v7i2.537.

[16] J. Kreps, N. Narkhede, and J. Rao, “Kafka: A Distributed Messaging System for Log Processing,” in *Proc. NetDB*, pp. 1–7, 2011. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf.

[17] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, 2003. Available: https://doi.org/10.1145/857076.857078.

[18] Amazon Web Services, “What is Event-Driven Architecture?,” *AWS Concepts*, 2024. Available: https://aws.amazon.com/event-driven-architecture/.

[19] Temporal Technologies, “Temporal Application Development Guide,” *Temporal Documentation*, 2024. Available: https://docs.temporal.io/dev-guide.

[20] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, D. Zhou, “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models,” *arXiv preprint arXiv:2201.11903*, vol. 35, pp. 1–43, 2022. Available: https://arxiv.org/abs/2201.11903.

[21] LangChain, “LangGraph: A Graph-Based Orchestration Framework,” *LangChain Blog*, 2024. Available: https://blog.langchain.dev/langgraph/.

[22] OpenTelemetry Authors, “OpenTelemetry Concepts: Distributed Tracing,” *OpenTelemetry.io*, 2024. Available: https://opentelemetry.io/docs/concepts/signals/traces/.

[23] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan C. Shanbhag, “Dapper, a Large-Scale Distributed Systems Tracing Infrastructure,” *Google Technical Report*, 2010. Available: https://research.google/pubs/pub36356/.

[24] H. Garcia-Molina and K. Salem, “Sagas,” in SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data, pp. 249–259, 1987. https://doi.org/10.1145/38713.38742.

[25] V. Vernon, “Implementing Domain-Driven Design”. *Addison-Wesley*, 2013. Available: https://books.google.co.id/books/about/Implementing_Domain_driven_Design.html?id=aVJsAQAAQBAJ.

[26] OpenAI, “Safety Best Practices,” *OpenAI Documentation*, 2024. Available: https://platform.openai.com/docs/guides/safety-best-practices.

[27] Y. Liu, G. Deng, Y. Li, K. Wang, Z. Wang, X. Wang, T. Zhang, Y. Liu, H. Wang, Y. Zheng, Y. Liu, “Prompt Injection Attack Against LLM-Integrated Applications,” *arXiv preprint arXiv:2306.05499*, 2023. Available: https://arxiv.org/abs/2306.05499.

[28] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-Free Replicated Data Types,” in *Proc. 13th Intl. Symp. Stabilization, Safety, and Security of Distributed Systems*, 2011. Available: https://hal.inria.fr/inria-00609399/document.

[29] J. Carlson, Redis in Action. *Manning Publications*, 2013. Available: https://www.manning.com/books/redis-in-action.

[30] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin, B. Ghanem, “CAMEL: Communicative Agents for ‘Mind’ Exploration of Large Language Model Society,” *arXiv preprint arXiv:2303.17760*, 2023. Available: https://arxiv.org/abs/2303.17760.

[31] K. Zhang, Z. Yang, and T. Başar, “Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms,” in Handbook of Reinforcement Learning and Control, vol. 325, 2021. https://doi.org/10.1007/978-3-030-60990-0_12.

[32] P. Lewis, et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in *NIPS'20: Proceedings of the 34th International Conference on Neural Information Processing Systems*, pp. 9459–9474, 2020. Available: https://dl.acm.org/doi/abs/10.5555/3495724.3496517.

[33] R. M. Monarch, Human-in-the-Loop Machine Learning. *Manning Publications*, 2021. Available: https://www.manning.com/books/human-in-the-loop-machine-learning.

[34] M. Kleppmann, Designing Data-Intensive Applications. *O'Reilly Media*, 2017. Available: https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/.

[35] B. Gates, “The Age of AI Has Begun,” *GatesNotes*, 2023. Available: https://www.gatesnotes.com/The-Age-of-AI-Has-Begun.