Observable Agent Coordination in Distributed Systems: A Comparative Analysis of Architectural Approaches

Nils Theres
Master of Applied IT
Fontys University of Applied Sciences
Eindhoven
n.theres@student.fontys.nl

Abstract

The increasing complexity of distributed systems demands efficient coordination of autonomous agents while maintaining system observability. This research investigated three architectural approaches to observable agent coordination: a centralized coordinator design, a decentralized peer-to-peer system, and a hybrid solution which combines centralized task decomposition with distributed execution. The architectures were implemented as proof-of-concept prototypes to examine their structural characteristics, coordination patterns, and observability implications, supported by empirical observations. The centralized approach demonstrated predictable performance and high observability but potential bottlenecks. The decentralized architecture offered better scalability at the cost of increased coordination complexity and limited observability. The hybrid approach achieved a balance between these characteristics by combining centralized control with distributed execution and persistent messaging. The findings suggest that hybrid architectures offer a promising direction for observable agent coordination in distributed systems.

1 Introduction

The increasing complexity of modern software systems drives the adoption of distributed architectures, where computational tasks are spread across multiple interconnected nodes. While distributed systems can improve scalability, resilience, and resource utilization [1, 2], the effective coordination of autonomous agents within these systems remains a significant challenge [3]. Software agents, which are routines that operate autonomously with local environmental awareness and without centralized control, form the foundation of modern distributed systems [4]. This paper examines different architectural approaches to coordinating and observing the interactions of these agents by focusing on the balance between coordination efficiency and system observability.

The challenge of maintaining observability in distributed agent systems is multifaceted and grows with system complexity. Observability is the ability to understand a system's internal state through its external outputs and it becomes particularly challenging in distributed environments [5]. System state becomes scattered across multiple autonomous agents which makes it difficult to maintain a coherent global view without introducing significant coordination overhead. This challenge is compounded by the need to track complex interactions between agents, understand decision-making processes, and maintain system insights even as agents dynamically join or leave the system. Recent incidents in large-scale distributed systems highlight these challenges: the 2021 Facebook outage demonstrated

Fontys University of Applied Sciences, Master of Applied IT Eindhoven, 2024.

how lack of observability can hamper debugging efforts in distributed systems [6], while Amazon's 2017 S3 outage showed how interdependent services can create cascading failures that are difficult to trace without proper observability mechanisms [7].

Consider a real-world scenario in modern e-commerce systems, where a seemingly simple user request triggers a complex chain of distributed operations. When a customer places an order, the system must coordinate across inventory management, payment processing, fraud detection, shipping optimization, and customer notification services. Each of these components may be handled by specialized agents distributed across different geographical regions and data centers. The challenge extends beyond mere task distribution: system operators need to understand how decisions are made, track task progression, identify bottlenecks, and quickly diagnose issues when they arise [5]. For instance, if a payment fails, operators need visibility into whether the failure occurred during the payment processing itself, during the fraud check, or due to communication issues between services. This level of observability becomes crucial for maintaining system reliability and user trust.

To address these observability challenges in multi-agent systems, this paper implements and compares three distinct architectural patterns: centralized, decentralized, and hybrid approaches. Each pattern offers different trade-offs between coordination efficiency and system observability, which we evaluate through empirical analysis focusing on several key aspects:

1. Task decomposition and allocation patterns
2. Agent coordination mechanisms
3. System observability characteristics
4. Performance and scalability implications

To conduct this evaluation, we implemented three proof-of-concept (PoC) prototypes, each representing a different architectural approach to agent coordination. Our analysis considers both quantitative metrics and qualitative aspects, with particular attention to how architectural choices impact system observability at scale.

As distributed computing paradigms grow increasingly complex, understanding these architectural trade-offs becomes crucial for system designers and operators. Our research provides practical insights into the strengths and limitations of each approach to help inform architectural decisions in observable distributed systems.

Given these practical challenges and requirements, we begin by examining existing research across several key dimensions, focusing particularly on approaches in task decomposition and coordination that enable observability. We then present detailed implementations of each architecture, followed by our evaluation methodology and experimental results. The paper concludes with a comparative analysis and discussion of future research directions.

1.1 Literature review

The challenges of observable agent coordination span multiple research areas, each contributing crucial insights to the design of distributed systems. We examine these contributions to understand how different architectural approaches might address the observability challenges outlined above.

1.1.1 Agent discovery and capability assessment

Dynamic agent discovery and capability assessment form critical components of distributed agent systems. Ben Noureddine. et al. introduced mechanisms that utilize intelligent registries to dynamically discover and assess agent capabilities based on task requirements to enable optimized agent-task pairing in dynamic environments [8]. This work established fundamental patterns for dynamic resource allocation, though questions of scalability and reliability in large-scale deployments remain open

challenges. Recent advances in federated systems have further highlighted the importance of robust discovery mechanisms, particularly in environments where agent availability may be uncertain [1].

1.1.2 Task decomposition and dependency management

Task decomposition serves as a core aspect of managing complex workflows in distributed systems. The representation of tasks as directed acyclic graphs (DAGs) has emerged as a prevalent approach, enabling clear dependency mapping and efficient parallel execution. Dong et al. demonstrated the effectiveness of DAGs for structured task management in their work on multi-agent coordination [9], which showed significant improvements in coordination efficiency and system scalability. Building on this foundation, Dutta et al. explored recursive DAG decomposition techniques that further reduce execution time and system overhead [10]. These approaches have proven particularly valuable in systems where task dependencies exhibit complex hierarchical relationships.

1.1.3 Coordination mechanisms

The coordination of autonomous agents presents ongoing challenges in multi-agent systems, with various approaches offering different trade-offs between centralization and autonomy. While centralized coordination provides stronger consistency guarantees, decentralized approaches often offer better scalability and resilience. Decentralized auction-based approaches, as surveyed by Skaltsis et al. [11], have demonstrated particular promise in dynamic task allocation for spatially distributed systems. Hybrid approaches combining elements of both paradigms have also emerged, though their effectiveness often depends heavily on specific application requirements and deployment contexts [3].

1.1.4 System observability

Observability has become increasingly critical as distributed systems grow in complexity. Recent research emphasizes the importance of comprehensive visibility into task states, inter-agent communications, and decision-making processes. Reily et al. established key principles for observable multi-agent systems [12]to highlight how proper instrumentation can facilitate debugging and system optimization. Real-world incidents have further stressed the importance of observability, particularly in large-scale distributed systems where debugging and incident response become challenging without proper visibility [2].

1.2 Research gap and synthesis

Prior research has made significant advances in individual areas. It ranges from task decomposition to coordination mechanisms and monitoring approaches. However, several key challenges remain unaddressed: While observability has been studied extensively in infrastructure monitoring [12], its relationship to different coordination architectures remains unexplored. Although both centralized and decentralized coordination approaches have demonstrated benefits [11, 5], their impact on system observability has not been systematically evaluated.

This gap between coordination mechanisms and their observability implications is particularly relevant for modern distributed systems, where both efficient execution and system visibility are essential. We address this gap through empirical evaluation of three architectural approaches, examining how different coordination patterns impact system observability.

2 Methodology

This study employed a multifaceted methodology to integrate theoretical analysis with practical implementations for developing and evaluating a framework for distributed multi-agent systems. The methodology consisted of the following components:

2.1 PoC development

To test theoretical insights and validate proposed solutions, three proof-of-concept implementations were developed, each representing a different architectural approach. These PoCs build on ideas from Ben Noureddine. et al. in the context of agent discovery [8] and coordination mechanisms studied by Skaltsis et al. [11]. The implementations were supported by a common framework using directed acyclic graphs. DAGs are mathematical structures consisting of nodes connected by directional edges that never form cycles [13]. They provide a natural representation for task decomposition and dependency management, where nodes represent sub-tasks and edges represent dependencies between them.

2.1.1 Task decomposition and DAG representation

Within our implementation of a multi-agent system, each task T is represented by a set of sub-tasks $\{T_1, T_2, \ldots, T_n\}$, where:

- Each sub-task T_i is a node in the DAG.
- Edges between nodes represent dependencies, denoted as depends_on$(T_i) = \{T_{j_1}, T_{j_2}, \ldots\}$, where T_{j_k} must be completed before T_i can execute.
- The terminal sub-task T_n produces the final result of T by combining outputs of all dependent tasks.

The workflow for an individual sub-task is defined as:

$$O_i = A_i(I_i), \quad I_i = \{O_{j_1}, O_{j_2}, \ldots\}, \quad \forall T_{j_k} \in \text{depends\_on}(T_i),$$

where:

- A_i: The agent assigned to T_i.
- I_i: The set of outputs from all tasks $\{T_{j_1}, T_{j_2}, \ldots\}$ on which T_i depends.
- O_i: The output of T_i, passed to dependent tasks as input.

For the terminal sub-task T_n, the final result of the task T is:

$$T = O_n.$$

2.1.2 Example workflow with dependencies

Consider the task: *"Fetch today's weather information in Celsius and a random number and add both."*. This task is decomposed into the following sub-tasks and dependencies:

- $T_{1.1}$: Fetch today's weather information in Celsius. depends_on$(T_{1.1}) = \{\}$.
- $T_{1.2}$: Fetch a random number. depends_on$(T_{1.2}) = \{\}$.
- $T_{1.3}$: Calculate the sum of both numbers. depends_on$(T_{1.3}) = \{T_{1.1}, T_{1.2}\}$.

The DAG representation and execution can be expressed as:

$$O_{1.1} = A_{1.1}(\emptyset), \quad O_{1.2} = A_{1.2}(\emptyset),$$

$$O_{1.3} = A_{1.3}(O_{1.1}, O_{1.2}), \quad T = O_{1.3}.$$

Or visually, as follows:

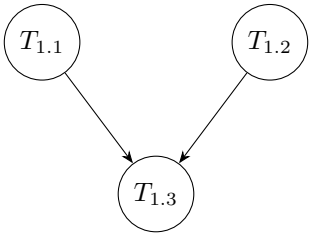

Figure 1: DAG representation of the task

The DAG ensures that:

1. $T_{1.1}$ and $T_{1.2}$ are executed in parallel as they have no dependencies.
2. $T_{1.3}$ is executed after both $T_{1.1}$ and $T_{1.2}$ are completed.
3. The terminal task $T_{1.3}$ signals the completion of the overall task T.

2.1.3 PoC 1: Centralized coordinator

PoC 1 (see Figure 2) employs a centralized coordinator architecture, where a single central entity is responsible for managing the entire workflow of a task. The process begins with the coordinator performing the initial decomposition of a task into smaller sub-tasks. These sub-tasks are then assigned to agents based on a capability assessment mechanism. The assessment evaluates the suitability of agents using task type compatibility.

Once an agent completes its assigned sub-task, the result is returned to the coordinator. The coordinator performs a dependency resolution check to verify whether prerequisites for subsequent sub-tasks have been fulfilled. If dependencies are resolved, additional sub-tasks are initiated. The workflow continues in this manner until the terminal sub-task is completed, at which point the task is marked as finished.

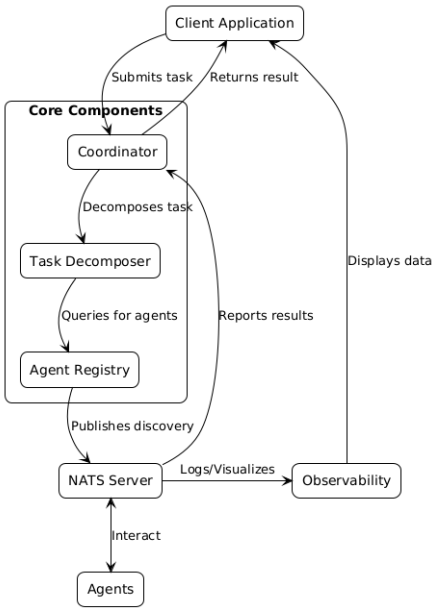

Figure 2: Centralized architecture of PoC 1

2.1.4 PoC 2: Decentralized architecture

PoC 2 (see Figure 3) introduces a fully decentralized architecture to emphasize autonomy and peer-to-peer collaboration among agents. In this model, all agents are equal by default, but they have the ability to be promoted to a leadership role for specific tasks. Leadership promotion follows a first-come-first-serve mechanism, where the first agent to successfully have its leadership claim acknowledged by another agent is designated as the leader for that task. The leader is responsible for task decomposition and the orchestration of sub-tasks.

Task assignment is based on a self-assessment mechanism, where agents evaluate their own capabilities against the requirements of sub-tasks broadcasted by the leader. Agents that qualify for a sub-task send claims to the leader, which acknowledges and finalizes the assignment.

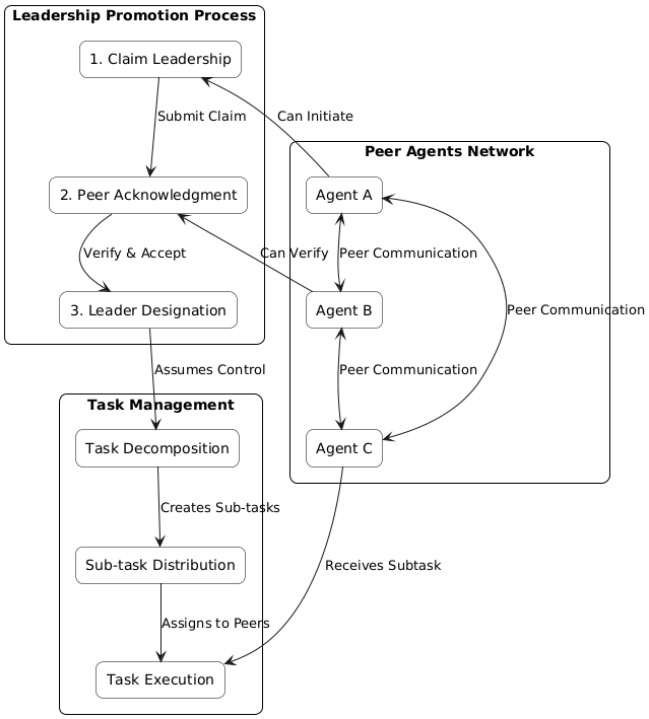

Figure 3: Decentralized architecture of PoC 2

2.1.5 PoC 3: Hybrid model

PoC 3 (see Figure 4) combines the strengths of centralized and decentralized architectures to offer a hybrid approach. A centralized coordinator is responsible for the initial task decomposition and the announcement of sub-tasks. Following this initial stage, agents self-assign sub-tasks based on their capabilities. Task delivery in this model is facilitated by delivery queues, which ensure exactly-once delivery semantics. Worker agents acknowledge task receipt through the queue. Specialized worker agents perform an introspection of the resolved internal DAG representation of the task to ensure that all dependencies are satisfied before processing begins.

The hybrid model is able to handle dynamic task creation at runtime. Worker agents can publish new sub-tasks to the broker to declare their dependencies and defer their original task until the new dependencies are resolved.

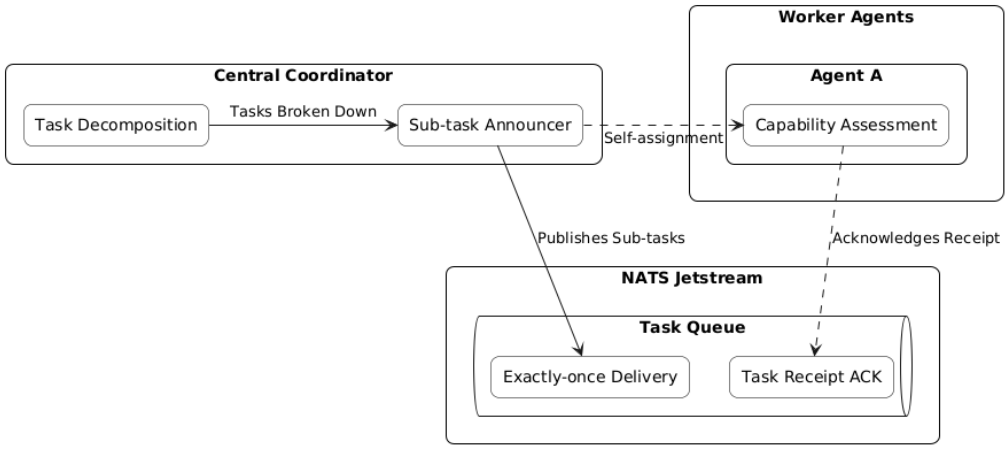

Figure 4: Hybrid architecture of PoC 3

2.2 Testing environment

The development and testing environment was established on a single development machine running all components. The system used a NATS[1] server (version 2.20.22) for message brokering, with the implementation performed in Python 3.12. Essential dependencies included the NATS Python client for message passing, JetStream[2] for persistent messaging in PoC 3, and AsyncIO for handling concurrent operations.

2.3 Benchmarking methodology

The benchmarking process was designed to evaluate the proposed solutions through both empirical measurements and theoretical analysis, with a focus on quantifiable metrics and architectural implications.

2.3.1 Empirically measured metrics

The empirical evaluation focused on four key areas of measurement. Task completion time was measured from the moment of task submission to its completion, similar to the performance metrics used in Dong et al.'s study of task dependencies [9]. Message patterns were analyzed to understand the nature and efficiency of inter-agent communication, building on ideas from the work of Malone and Crowston on coordination patterns in distributed systems [14]. The coordination overhead was assessed by measuring the time spent in coordination activities versus the actual execution of the task. Implementation complexity was evaluated through several factors, including the volume of code required for each architecture, the number of distinct message subjects, and coordination points. The latter refers to parts of the implementation that require explicit coordination efforts, such as task distribution or result announcement.

2.3.2 Test scenarios

Drawing from established experimental design practices outlined in distributed systems research [1], the benchmark testing utilized a two-step task workflow with dependencies. To ensure statistical significance, each test was repeated through 100 iterations. The testing involved a configuration of two to three agents per architecture to allow for basic interaction patterns.

[1] https://nats.io/
[2] https://docs.nats.io/nats-concepts/jetstream

2.3.3 Theoretical analysis

Given the constraints of the testing environment, several aspects required theoretical analysis rather than empirical measurement. The scalability analysis, inspired by the ISO/IEC 25002:2024 standard for software quality evaluation[3], examined architectural patterns and their implications for larger deployments. Fault tolerance was assessed through a theoretical evaluation of potential failure modes and recovery mechanisms and was also guided by the principles outlined in the aforementioned ISO standard.

2.3.4 Observability assessment

Our observability assessment methodology draws ideas from the work of Burgess [5] and evaluates each architecture through three key features: task state visibility, decision traceability, and debug capability.

Task state visibility was rated as "Complete" when all task state transitions were observable and logged with full context, and "Partial" when state information was fragmented or required correlation across multiple sources.

Decision traceability was assessed as "High" when the full chain of decision-making events could be reconstructed from logs and traces, "Limited" when gaps existed in the decision trail or when certain decisions could not be fully explained from available data.

Debug capability was rated "High" when the system provided comprehensive logging, clear error messages, and the ability to correlate events across components. A "Medium" rating indicated partial debug information or challenges in correlating events across system components.

These assessments were based on examining log completeness, state change granularity, inter-agent communication visibility, and our ability to reconstruct event sequences during testing.

3 Results

The evaluation of the three architectural approaches revealed distinct characteristics in terms of performance, complexity, and behavioral patterns. This section presents the empirical results of our implementation and testing, followed by a comparative analysis of the architectures.

3.1 Implementation complexity

Analysis of implementation complexity reveals differences between the three architectures. Table 1 presents the metrics related to the effort and complexity of the implementation.

Metric	PoC 1	PoC 2	PoC 3
Lines of code	195	205	204
Number of message subjects	3	9	3
Coordination Points	4	9	5

Table 1: Implementation complexity metrics

The recorded message counts varied significantly between implementations. PoC 1 utilized four messages per task execution, comprising coordinator publications and completion notifications. PoC 2 operated with nine distinct message subjects and recorded 18 messages per task execution. PoC 3 showed six messages per task execution, consisting of history stream events, subtask publications, and result publications.

[3] https://www.iso.org/standard/78175.html

3.2 Performance characteristics

Performance testing revealed distinct behavioral patterns across the three architectures. The evaluation focused on three key aspects: task completion efficiency, coordination overhead and observability.

3.2.1 Task completion time

The centralized coordinator (Figure 5, PoC 1) demonstrated consistent task completion times with low variance (±2.4 ms std. deviation). The decentralized architecture (Figure 5, PoC 2) showed slightly higher average completion times (±4.6 ms std. deviation). The hybrid approach (Figure 5, PoC 3) achieved completion times that perform between these two data points (±4.4 ms std. deviation).

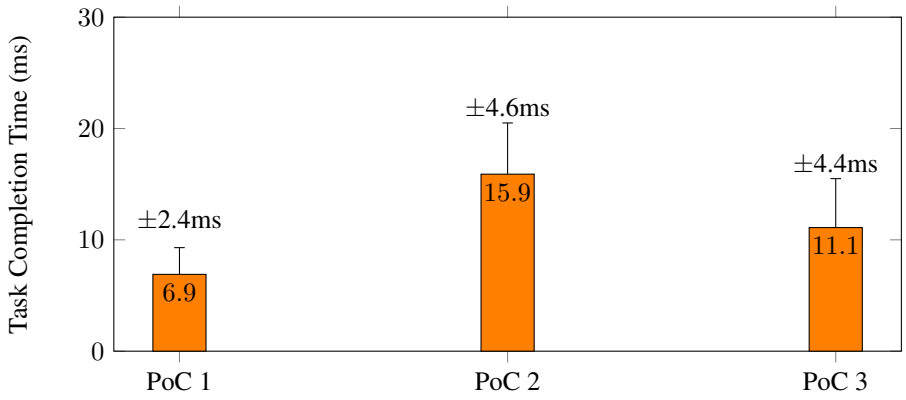

Figure 5: Task completion times with variances

3.2.2 Coordination overhead

Analysis of coordination overhead revealed interesting patterns across the implementations.

Metric	PoC 1	PoC 2	PoC 3
Average Coordination Time (ms)	4.635	1.583	1.032
Message Count per Task	4	18	6
Leader Election Time (ms)	N/A	6.502	N/A
Dependency Resolution Time (ms)	0.010	0.001	0.593

Table 2: Coordination overhead analysis

The highest measured values were 18 messages per task and a coordination time of 4.635 ms (see Table 2). Leader election was only present in PoC 2, adding 6.502 ms to the coordination process.

3.3 Observability analysis

The three architectures demonstrated varying levels of system observability. PoC 2 exhibited the most notable differences (see Table 3), with partial task state visibility, limited decision traceability, and medium debug capability, contrasting with the complete task state visibility, high decision traceability, and high debug capability observed in PoC 1 and PoC 3.

Observability Feature	PoC 1	PoC 2	PoC 3
Task State Visibility	Complete	Partial	Complete
Decision Traceability	High	Limited	High
Debug Capability	High	Medium	High

Table 3: Observability metrics comparison

4 Discussion

The analysis of the three architectural approaches revealed distinct patterns in coordination, observability, and performance characteristics. Each architecture demonstrates specific trade-offs that make it suitable for different use cases.

4.1 Architectural patterns and performance characteristics

PoC 1's centralized architecture implements a classical star topology, where all communication flows through a central coordinator [15]. This pattern, while simple to implement and reason about, creates a potential bottleneck at the coordinator. However, it simplifies synchronization through its single point of control, which eliminates the need for complex protocols [15]. The centralized nature enables efficient dependency resolution (0.010ms) since all task state and dependency information is locally available to the coordinator. This centralization of communication flow also creates natural system observation points. Each message passing through the coordinator provides precise task state information and progression tracking without requiring additional monitoring mechanisms. This architectural choice leads to consistent but capacity-limited performance (6.9ms ±2.4ms), demonstrating how simplified synchronization can provide predictability at the cost of scalability.

PoC 2's decentralized approach employs a peer-to-peer pattern with dynamic leadership. This requires an elaborate synchronization protocol [15]. The architecture demands nine distinct message types to manage its synchronization flow:

1. Leadership establishment requiring lock acquisition and acknowledgment
2. Task claim competition among agents
3. Formal task assignment with acknowledgment
4. Synchronized task kickoff

Each agent maintains its own in-memory record of dependencies. While this results in fast dependency resolution (0.001ms), it requires extensive synchronization to prevent race conditions. A particular challenge arises in result handling: without persistent storage, late-joining workers might miss results published before their subscription, potentially leading to deadlocks in dependent tasks. This complexity is reflected in the high message count (18 messages per task) and significant leader election overhead (6.502ms, see Figure 2). The distributed nature of state information makes system observation more challenging. No single agent has a complete view of the system state, which requires additional coordination to maintain coherent system visibility. These architectural choices result in reduced performance with higher variability (15.9ms ±4.6ms), reflecting the overhead costs of decentralized coordination, though the approach offers better theoretical scalability through the elimination of central bottlenecks.

PoC 3's hybrid architecture bears striking similarities to modern CPU designs. Like a CPU's control unit managing distributed execution units, the central coordinator handles task distribution while worker agents perform parallel execution [16]. The JetStream component acts similarly to a CPU's message bus, providing guaranteed delivery semantics. By using delivery queues with exactly-once semantics and a persistent history stream, the architecture eliminates many of the race

conditions and timing issues present in PoC 2. While this introduces slower dependency resolutions (0.593ms) due to key-value store queries, it significantly reduces the complexity of task coordination and result handling. Results are durably stored in the history stream to allow workers to access them regardless of subscription timing. This persistent message history serves a dual purpose: it ensures reliable task coordination while simultaneously providing a complete audit trail for system observation, effectively decoupling observation capabilities from real-time message flow. This balanced architectural approach achieves moderate performance characteristics (11.1ms ±4.4ms) and demonstrates how persistent messaging can provide predictable behavior while maintaining scaling capabilities.

4.2 Fault tolerance potential

Although fault tolerance mechanisms were not implemented in the current prototypes, the architectural patterns of each approach suggest different capabilities and challenges for handling system failures, particularly in maintaining system observability during failure scenarios.

The centralized architecture of PoC 1 presents a clear single point of failure in its coordinator. While this simplifies failure detection, as all agents report to a single entity, it also means that coordinator failure would halt the entire system. This centralization creates a tight coupling between system operation and system observation. Coordinator failure would disable both task execution and the ability to monitor system state. Recovery mechanisms could be implemented through coordinator redundancy, though this would require additional complexity to maintain state consistency between primary and backup coordinators. The consistent task completion times observed in our results (see Figure 5) suggest this architecture could enable rapid failure detection through timing violations, as deviations from these stable patterns would be immediately apparent to the coordinator.

PoC 2's decentralized architecture inherently avoids single points of failure, as any agent can potentially assume leadership. The existing leader election mechanism provides a foundation for fault recovery, as similar processes could be used to elect new leaders when failures occur. However, the lack of persistent storage means that task state and results could be lost during agent failures. This would require complex state replication protocols between peers to ensure reliability. The distributed nature of the system creates additional challenges for failure observation. With only partial state visibility at each node, distinguishing between node failures and network partitions becomes more complex. The higher variance in task completion times (see Figure 5) further complicates failure detection through timing analysis alone.

PoC 3's hybrid approach is particularly promising for fault tolerance implementation. JetStream's persistent message storage provides built-in failure recovery capabilities. Failed workers can be replaced without losing task state or results, as all necessary information is preserved in the message history and key-value store. The centralized coordinator could be made fault-tolerant through similar mechanisms to PoC 1, while maintaining the ability to recover worker state through JetStream's persistence layer. Task acknowledgments and exactly-once delivery semantics further ensure that no work is lost during component failures. Most importantly, the architecture maintains observability even during failure scenarios. The message history provides a consistent view of system state independent of individual component availability.

4.3 Observability implications

Each architecture presents different observability characteristics that directly impact system maintainability and debugging. The centralized design of PoC 1 provides complete task state visibility and high decision traceability through its single point of control. In contrast, PoC 2's peer-to-peer nature makes maintaining a coherent system view challenging, with only partial state visibility and limited decision traceability.

The hybrid approach of PoC 3 combines the best of both worlds. Similar to how modern CPUs provide performance counters and debug registers [17], the combination of centralized task decomposition

and JetStream's built-in tracing provides comprehensive system observability without sacrificing distributed execution benefits.

5 Limitations

Several significant limitations influenced the methodology and scope of testing. The single-machine testing environment limited the ability to assess true distributed performance characteristics. Scale limitations prevented testing with large numbers of agents or high message volumes. The testing environment also constrained the ability to simulate realistic failure scenarios effectively. Additionally, the shared environment made it difficult to isolate and measure resource usage per component.

The methodology was also limited by the use of a single test scenario type (two-step workflow) and the absence of real-world workload patterns. Implementation choices, such as the exclusive use of Python and NATS message broker, may affect the generalizability of results to systems using different technology stacks. Finally, while timing metrics were thoroughly evaluated, the study was unable to assess long-term reliability characteristics or behavior under different network topologies that would be present in geographically distributed deployments.

6 Ethical considerations

Although this research focuses primarily on technical implementation, ethical considerations are important to acknowledge. The research prioritized transparency in system decision-making processes by designing architectures that make agent behaviors and interactions observable and understandable, a critical requirement as distributed systems increasingly impact critical infrastructure and services. Each architectural approach was evaluated for its ability to provide accountability through comprehensive tracking and tracing capabilities.

The hybrid architecture's persistent messaging system, while designed primarily for reliability, has implications for audit capabilities by guaranteeing that system behaviors can be reconstructed and verified. This characteristic becomes particularly relevant as distributed systems are deployed in regulated environments where accountability is essential.

Future work should investigate security implications and potential misuse scenarios in observable agent architectures more thoroughly, particularly regarding unauthorized monitoring and the privacy implications of comprehensive system logging.

7 Conclusion

This paper introduced and evaluated three distinct approaches to observable agent coordination in distributed systems. Through empirical evaluation and analysis, each architecture demonstrated unique characteristics and trade-offs in terms of complexity, performance, and observability.

The centralized architecture demonstrated that while single-point coordination simplifies system observation and dependency management, it creates fundamental scalability limitations. The decentralized approach showed that removing central coordination enables better scalability but at the cost of significantly increased complexity in both implementation and system observation. The hybrid architecture revealed that persistent messaging can effectively decouple system observability from coordination mechanisms to enable scalable operation while maintaining comprehensive system visibility.

Our analysis revealed a fundamental relationship between coordination patterns and observability capabilities. Systems optimized purely for coordination efficiency tend to sacrifice observability, while those optimized for observability often introduce coordination overhead. The hybrid approach suggests that this trade-off can be mitigated through architectural choices that separate observability concerns from core system operation.

Several limitations in our current evaluation suggest promising directions for future research. Primary among these is the need to evaluate these architectures in true distributed environments with larger agent populations and real-world network conditions. Future work should also investigate security implications and potential misuse scenarios, particularly regarding unauthorized monitoring and privacy implications of comprehensive system logging. Additional research opportunities include developing more sophisticated failure recovery mechanisms and exploring performance optimizations for complex task dependencies.

The findings suggest that while each architecture has its merits, the hybrid approach offers a promising direction for observable agent coordination in distributed systems. By treating observability as a distinct architectural concern rather than a byproduct of coordination mechanisms, systems can achieve better scalability without sacrificing the ability to understand and debug their behavior. As distributed systems continue to grow in complexity and scale, this balanced approach to technical capabilities and accountability requirements will become increasingly important for building reliable and trustworthy systems.

References

[1] Q. Li et al. “A Survey on Federated Learning Systems: Vision, Hype and Reality for Data Privacy and Protection”. In: *IEEE Transactions on Knowledge and Data Engineering* 35 (2019), pp. 3347–3366. DOI: 10.1109/TKDE.2021.3124599.

[2] Yajing Xu et al. “BESIFL: Blockchain-Empowered Secure and Incentive Federated Learning Paradigm in IoT”. In: *IEEE Internet of Things Journal* 10 (2023), pp. 6561–6573. DOI: 10.1109/JIOT.2021.3138693.

[3] Jianrui Wang et al. “Cooperative and Competitive Multi-Agent Systems: From Optimization to Games”. In: *IEEE/CAA Journal of Automatica Sinica* 9 (2022), pp. 763–783. DOI: 10.1109/JAS.2022.105506.

[4] Michael J. Wooldridge. *An Introduction to Multiagent Systems*. 2nd ed. Chichester: John Wiley & Sons, 2009. ISBN: 978-0-470-51946-2.

[5] Mark Burgess. *From Observability to Significance in Distributed Information Systems*. July 25, 2019. DOI: 10.48550/arXiv.1907.05636. arXiv: 1907.05636 [cs]. URL: http://arxiv.org/abs/1907.05636 (visited on 12/27/2024). Pre-published.

[6] “Millions Flock to Signal and Telegram After Facebook Outage”. In: *Bloomberg.com* (Oct. 5, 2021). URL: https://www.bloomberg.com/news/articles/2021-10-05/millions-flock-to-signal-as-facebook-whatsapp-suffer-outage (visited on 12/14/2024).

[7] *Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region*. https://aws.amazon.com/message/41926/. (Visited on 12/14/2024).

[8] Dhouha Ben Noureddine. et al. “Multi-agent Deep Reinforcement Learning for Task Allocation in Dynamic Environment”. In: *Proceedings of the 12th International Conference on Software Technologies - ICSOFT*. INSTICC. SciTePress, 2017, pp. 17–26. ISBN: 978-989-758-262-2. DOI: 10.5220/0006393400170026.

[9] Yubo Dong et al. “VillagerAgent: A Graph-Based Multi-Agent Framework for Coordinating Complex Task Dependencies in Minecraft”. In: *Findings of the Association for Computational Linguistics: ACL 2024*. Ed. by Lun-Wei Ku et al. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 16290–16314. DOI: 10.18653/v1/2024.findings-acl.964. URL: https://aclanthology.org/2024.findings-acl.964.

[10] Anushree Dutta et al. “Divide-and-Conquer-Based Recursive Decomposition of Directed Acyclic Graph”. In: *Soft Computing Techniques and Applications*. Ed. by Samarjeet Borah et al. Singapore: Springer Singapore, 2021, pp. 305–320. ISBN: 978-981-15-7394-1.

[11] George Marios Skaltsis et al. “A survey of task allocation techniques in MAS”. In: *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*. June 2021, pp. 488–497. DOI: 10.1109/ICUAS51884.2021.9476736.

[12] Brian Reily et al. “Role Discovery in Observed Multi-Agent Systems Over Time through Matrix Factorization”. In: *2021 International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*. 2021, pp. 66–74. DOI: 10.1109/MRS50823.2021.9620581.

[13] Jørgen Bang-Jensen and Gregory Gutin, eds. *Classes of Directed Graphs*. Springer Monographs in Mathematics. Cham: Springer International Publishing, 2018. ISBN: 978-3-319-71839-2 978-3-319-71840-8. DOI: 10.1007/978-3-319-71840-8. (Visited on 01/13/2025).

[14] Thomas W. Malone and Kevin Crowston. “The Interdisciplinary Study of Coordination”. In: *ACM Comput. Surv.* 26.1 (Mar. 1994), pp. 87–119. ISSN: 0360-0300. DOI: 10.1145/174666.174668. (Visited on 01/14/2025).

[15] Naomi J. Alpern and Robert J. Shimonski. “CHAPTER 1 - Network Fundamentals”. In: *Eleventh Hour Network+*. Ed. by Naomi J. Alpern and Robert J. Shimonski. Boston: Syngress, Jan. 1, 2010, pp. 1–18. ISBN: 978-1-59749-428-1. DOI: 10.1016/B978-1-59749-428-1.00003-5. URL: https://www.sciencedirect.com/science/article/pii/B9781597494281000035 (visited on 12/27/2024).

[16] R. Wiśniewski. *Synthesis of compositional microprogram control units for programmable devices*. Vol. 14. Lecture Notes in Control and Computer Science. Zielona Gora: University of Zielona Gora Press, 2009, p. 153. ISBN: 978-83-7481-293-1.

[17] Bernard Goossens. *Guide to Computer Processor Architecture: A RISC-V Approach, with High-Level Synthesis*. Undergraduate Topics in Computer Science. Cham: Springer International Publishing, 2023. ISBN: 978-3-031-18022-4 978-3-031-18023-1. DOI: 10.1007/978-3-031-18023-1. URL: https://link.springer.com/10.1007/978-3-031-18023-1 (visited on 12/27/2024).