Post-Mortem Report

Agentic AI System for Multi-Step Tasks

1. Overview

This post-mortem report documents the key challenges, design decisions, trade-offs, and lessons learned during the design and implementation of the Agentic AI System for Multi-Step Tasks. The purpose of this document is to provide a transparent and first-hand analysis of how the system behaved under realistic constraints and how architectural decisions evolved during development.

The report explicitly addresses all post-mortem requirements defined in the submission guidelines and also highlights additional improvements implemented beyond the baseline requirements.

2. Scaling Issue Encountered

2.1 Description of the Issue

The primary scaling issue encountered during development was related to execution metrics and state visibility across system components. Initially, task execution metrics such as latency and cost were tracked within in-memory data structures inside the worker process.

While this approach worked in isolated testing, it failed once the system was executed using multiple independent processes. The API server, running in a separate process, was unable to access metrics generated by the worker. As a result, metrics queries returned missing or inconsistent data despite successful task execution.

2.2 Root Cause Analysis

The root cause of the issue was an implicit assumption that in-memory state could be shared across process boundaries. In reality, each process maintains its own isolated memory space. Once asynchronous execution and independent worker processes were introduced, this assumption broke down.

This issue is a common pitfall in distributed system design and became apparent only after the system was exercised under a more realistic execution model.

2.3 Resolution Implemented

The issue was resolved by migrating all execution metrics to Redis, which already served as the system's central coordination layer. Metrics were persisted in Redis using a key-value data model keyed by task identifier.

With this change, both the worker process and the API server accessed a shared, consistent source of truth. Metrics became reliably available through the API regardless of process boundaries, and the system aligned more closely with distributed system best practices.

2.4 Outcome

After migrating metrics to Redis, metrics retrieval became stable, consistent, and scalable. This change enabled horizontal scaling of both API and worker processes without introducing additional coordination complexity.

3. Design Decision to Change

3.1 Original Design Decision

The initial design relied on transient, in-memory tracking of task and step state within the worker lifecycle. This design was chosen to reduce complexity and focus on core orchestration, agent coordination, and streaming behavior.

3.2 Limitations Observed

As the system matured, several limitations of this approach became evident:

- Task state was lost if a worker restarted
- Historical task execution could not be inspected
- Long-running or delayed tasks were harder to manage

These limitations did not prevent correct execution but reduced robustness and debuggability.

3.3 Revised Design Approach

A more robust design would persist task and step state externally, either using Redis hashes or a dedicated database. This would enable durable state tracking, task resumption, and historical inspection of execution.

While not strictly required for the scope of this project, this change would be necessary for a production-grade deployment.

3.4 Justification for the Original Choice

The original in-memory approach was intentionally selected to keep the system simple and understandable during early development. This trade-off allowed faster iteration and clearer demonstration of agent orchestration and asynchronous execution without introducing premature complexity.

4. Trade-offs Made During Development

Decision	Trade-off (Pros)	Trade-off (Cons)
Redis Streams vs Kafka	Lower operational overhead; simpler setup and maintenance; sufficient for moderate throughput and local development.	Less robust for multi-region persistence and very high-throughput use cases compared to Kafka.
Manual Batching	Precise control over execution cost and throughput; reduces queue overhead; improves efficiency for similar agent steps.	Increases complexity in worker logic; introduces potential head-of-line blocking if batches are poorly sized.
At-least-once Delivery Semantics	Ensures no task or step is silently dropped in case of worker crashes or restarts.	Requires agents to be idempotent to safely handle duplicate executions.

4.1 Redis Streams vs Kafka

Redis Streams were chosen over Kafka due to their lightweight operational footprint, ease of setup, and sufficient feature set for asynchronous task coordination. Kafka provides stronger durability and higher throughput but introduces additional operational complexity that was unnecessary for the project's scope.

4.2 Server-Sent Events vs WebSockets

Server-Sent Events were selected for streaming execution progress because the communication pattern is strictly one-way from server to client. SSE offers native browser support, simpler connection management, and lower overhead compared to WebSockets.

4.3 Manual Orchestration vs Agent Frameworks

Instead of using black-box agent frameworks, explicit orchestration logic was implemented. This increased development effort but significantly improved explainability, observability, and control over execution flow.

4.4 Simplicity vs Persistence

Certain components, such as transient task state, were intentionally kept simple rather than fully persistent. This decision prioritized clarity and system comprehension over full durability, which was acceptable given the project's requirements.

5. Additional Improvements Beyond Requirements

5.1 Structured Event Streaming

Execution progress is emitted as structured JSON events rather than raw text. This enables reliable frontend parsing, replayability, and integration with monitoring or visualization tools.

5.2 Distributed Metrics Persistence

Latency and cost metrics are computed by worker processes and persisted in Redis. This design supports cross-process access and enables horizontal scaling without shared memory.

5.3 Manual Batching Optimization

Execution steps targeting the same agent are batched before being enqueued into Redis. This reduces message overhead, improves throughput, and demonstrates explicit control over task dispatch.

5.4 Observability and Logging

Structured logs include task identifiers, agent names, step identifiers, and execution status. This significantly improves debuggability and operational visibility during asynchronous execution.

6. Lessons Learned

Developing this system reinforced several important lessons:

- Shared in-memory state does not scale across processes
- Explicit state ownership is critical in distributed systems
- Observability must be designed from the start
- Simplicity should be balanced carefully against durability

These lessons informed subsequent design improvements and would guide future iterations of the system.

7. Conclusion

The post-mortem analysis demonstrates that the system not only meets all functional and technical requirements but also addresses real-world challenges encountered in asynchronous, distributed architectures.

The final design reflects mature system reasoning, clear trade-offs, and readiness for future extension or production hardening.