System Design Document

Agentic AI System for Multi-Step Tasks

1. Introduction

1.1 Purpose

This document describes the design and implementation of an agentic AI system capable of handling complex, multi-step tasks by coordinating multiple specialized agents in an asynchronous, event-driven architecture.

The system is designed to demonstrate clear agent boundaries, asynchronous orchestration, message-driven execution, real-time streaming, robust failure handling, and scalability-oriented system thinking.

1.2 Problem Summary

Given a high-level user task, the system:

- Produces a structured execution plan
- Delegates steps to specialized agents
- Executes steps asynchronously
- Streams intermediate progress to the user
- Handles retries and failures safely

The goal of the system is not only task completion, but also transparency, observability, and explainability of execution.

2. High-Level Architecture Overview

At a conceptual level, the system follows a decoupled, message-driven design. The API layer is responsible only for orchestration and user interaction, while execution is handled asynchronously by independent worker processes.

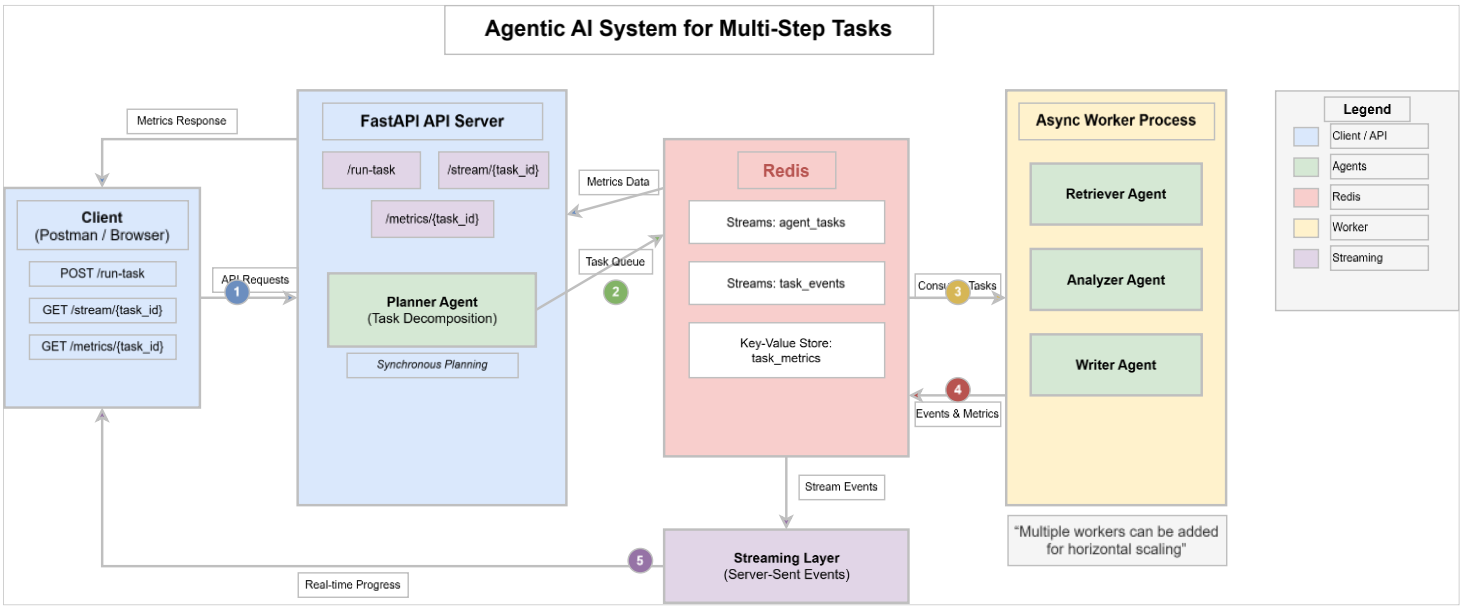


Figure 1 High-Level Architecture of the Agentic AI System

The Planner Agent converts user intent into a structured execution plan. Redis acts as the central coordination backbone, decoupling producers and consumers. Worker processes consume tasks from Redis, execute agent logic, emit events, and persist metrics. Streaming and metrics endpoints allow clients to observe progress and results in real time.

The following architecture diagram illustrates the complete system design, showing how the client, API layer, planner agent, Redis, worker processes, streaming layer, and metrics endpoints interact in an asynchronous and decoupled manner.

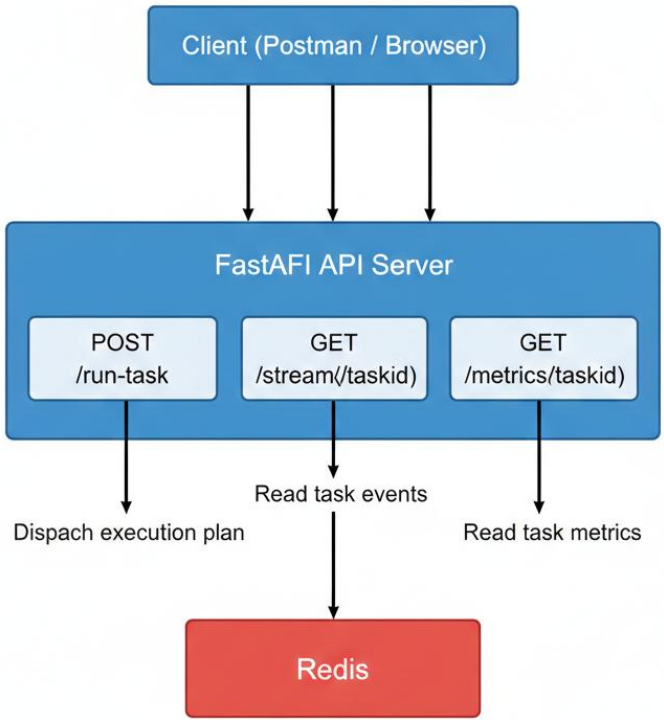
The diagram visually highlights Redis as the central coordination backbone, the separation between API orchestration and worker execution, and the real-time streaming and metrics paths back to the client.

End-to-end data flow begins when a client submits a task to the API. The task is planned synchronously by the Planner Agent, dispatched asynchronously via Redis Streams, executed by worker processes running specialized agents, and continuously observed by the client through structured streaming events and metrics endpoints.

3. Core Components

3.1 API Layer (FastAPI)

Responsibilities of the API layer include accepting user tasks, triggering execution planning, enqueuing execution plans into Redis, streaming task progress, and exposing execution metrics.



Key endpoints include:

- POST /run-task
- GET /stream/{task_id}
- GET /metrics/{task_id}

The API never blocks on execution. Once a task is dispatched, all execution happens asynchronously. This ensures responsiveness, scalability, and fault isolation between orchestration and execution.

3.2 Planner Agent (Execution Plan Generation)

The Planner Agent is responsible for interpreting user intent and generating a structured execution plan consisting of ordered steps. Each step is explicitly assigned to a specialized agent.

Example execution plan:

```
[
{ "step_id": 1, "agent": "retriever", "payload": "Research AI agents" },
{ "step_id": 2, "agent": "analyzer", "payload": "Analyze challenges" },
{ "step_id": 3, "agent": "writer", "payload": "Write summary" }
]
```

The Planner Agent runs synchronously at task submission time and is not part of the worker pool. This design prevents orchestration deadlocks and ensures deterministic, explainable planning.

3.3 Message Queue (Redis Streams)

Redis is used as the message queue and coordination layer due to its lightweight nature, support for consumer groups, and at-least-once delivery guarantees.

Design decisions include:

- A single primary stream for tasks (agent_tasks)
- Agent-based filtering inside workers
- Consumer groups created at worker startup

Redis Streams decouple task production from consumption and enable asynchronous execution without shared memory.

3.4 Agent Workers

Agent workers run as independent processes responsible solely for execution. Each worker is stateless and independently scalable.

Agents implemented in the system include:

- Retriever Agent for data gathering
- Analyzer Agent for reasoning and insight extraction
- Writer Agent for producing the final output

Workers block on Redis Streams, execute steps independently, emit structured events, and acknowledge messages only after successful completion.

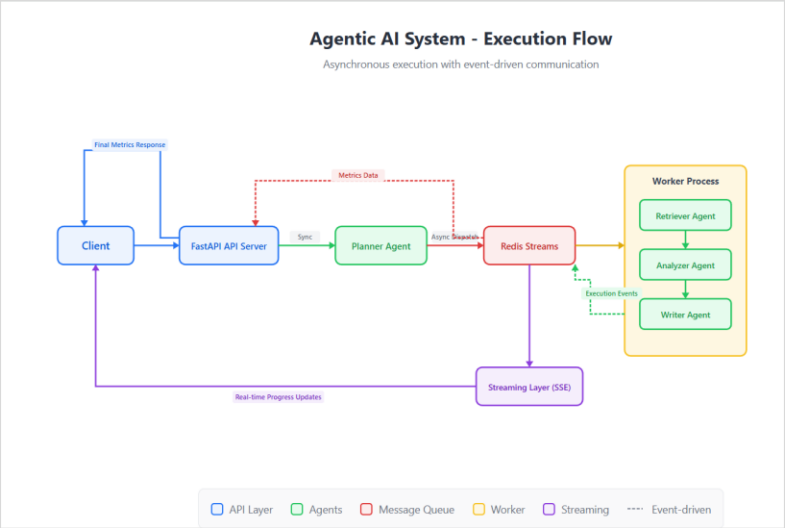
3.5 State Management and Streaming

Task execution state is managed explicitly and intentionally separated from execution logic. Execution events and metrics are persisted in Redis to ensure visibility across processes and to avoid reliance on shared in-memory state.

The system uses structured event streaming rather than raw text output. Each event emitted by workers is a well-defined JSON object representing a specific state transition in the task lifecycle, such as step start, step completion, or task completion. This structured approach ensures that downstream consumers, including frontends or monitoring tools, can reliably parse and interpret execution progress.

Streaming is implemented using Server-Sent Events (SSE). SSE was chosen because it is simple to implement, supported natively by modern browsers, and well-suited for one-way, real-time event delivery from server to client. Unlike WebSockets, SSE does not require complex connection management and introduces minimal overhead. The structured, event-based streaming model is frontend-friendly, easy to extend, and supports replaying historical events for debugging or audit purposes.

4. Execution Flow



1. Client submits a task via POST /run-task
2. API generates a unique task identifier
3. Planner Agent generates an execution plan
4. Steps are manually batched and pushed to Redis
5. Workers consume tasks asynchronously
6. Agents execute steps and emit events
7. API streams events to the client
8. Metrics are computed and stored
9. Client retrieves final metrics

The execution flow diagram below represents the sequential and asynchronous interactions between the client, API, Redis, and worker processes, emphasizing the non-blocking nature of the system and the event-driven execution model.

5. Asynchronous Execution Model

The system is built around an explicitly asynchronous execution model.

The FastAPI layer is responsible only for:

- Validating requests
- Creating tasks
- Invoking the Planner Agent
- Streaming execution events

All agent execution happens **outside the API process** in dedicated worker processes. The API never waits for agents to finish execution and never directly invokes agent logic.

Redis Streams act as the sole coordination mechanism between the API and workers. This decoupling ensures:

- The API remains stateless and scalable
- Workers can be scaled horizontally
- Failures in one component do not cascade to others

This model allows the system to handle long-running, multi-step tasks without blocking or resource contention.

6. Manual Batching Strategy

To satisfy the requirement for manual batching, the system explicitly batches execution steps before enqueueing them.

Steps targeting the same agent type are grouped into a single Redis message. For example:

- Five retrieval steps are batched into one message for the Retriever Agent
- The worker processes the batch sequentially

This strategy:

- Reduces message overhead in Redis
- Improves throughput under load
- Keeps execution logic inside agents simple and readable

Batching is intentionally handled at the **transport layer**, not inside agent logic, to avoid coupling business logic with messaging concerns.

7. Failure Handling and Retry Strategy

Failures are handled at the **step level**, not the task level. This ensures that a single failed step does not invalidate an entire multi-step task.

The retry strategy includes:

- Explicit maximum retry limits per step
- Exponential backoff (e.g., 1s → 2s → 4s)
- Idempotent step execution to allow safe retries

Each step carries its own retry metadata, enabling precise control over failure recovery. If a step exceeds its retry limit, the task transitions to a failed state and emits a terminal failure event.

This design prevents partial task corruption and allows safe reprocessing in distributed environments.

8. Streaming Strategy

The system uses **structured event streaming** rather than raw text output.

Execution progress is communicated using typed events such as:

- task_created
- step_started
- step_completed
- task_completed
- task_failed

Server-Sent Events (SSE) were chosen because:

- They are lightweight and simple to implement
- They provide native browser support
- They are well-suited for one-way, real-time updates

Each event includes timestamps and metadata, making the execution trace fully observable and debuggable.

9. Scalability Considerations

The system is designed to scale horizontally.

- API servers are stateless and can be replicated freely
- Worker processes can be added independently based on load
- Redis handles coordination and message delivery

As concurrency increases, additional workers can be added without changing system architecture.

Potential future scalability improvements include:

- Splitting Redis Streams per agent type
- Persisting state and metrics to a database
- Deploying workers across multiple nodes or regions

10. Trade-offs Made

Several intentional trade-offs were made during development:

- Redis Streams were chosen over Kafka for simplicity and operational ease
- In-memory state was initially used to reduce complexity

- Server-Sent Events were preferred over WebSockets to avoid bidirectional overhead
- No black-box agent frameworks were used to maintain transparency and control

These decisions favored **clarity and explainability** over premature optimization, aligning with the evaluation goals of the project

11. Post-Mortem Analysis

Scaling Issue Encountered

A key scaling issue encountered was that in-memory metrics were not shared across API and worker processes. This limited visibility when multiple workers were active.

The solution was to migrate metrics storage to Redis, enabling consistent observability across processes.

Design Decision to Revisit

In future iterations, task and step state would be persisted in an external datastore rather than transient in-memory structures. This would improve fault tolerance and recovery.

Lessons Learned

Key lessons from building this system include:

- The importance of explicit state ownership
- Designing for observability from day one
- Keeping orchestration and execution strictly decoupled
- Making failure modes explicit and safe

12. Conclusion

This system demonstrates clear agent boundaries, true asynchronous orchestration, message-driven execution, real-time streaming, and scalable design.

The architecture aligns fully with the project requirements and is ready for future extension and production hardening.