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.

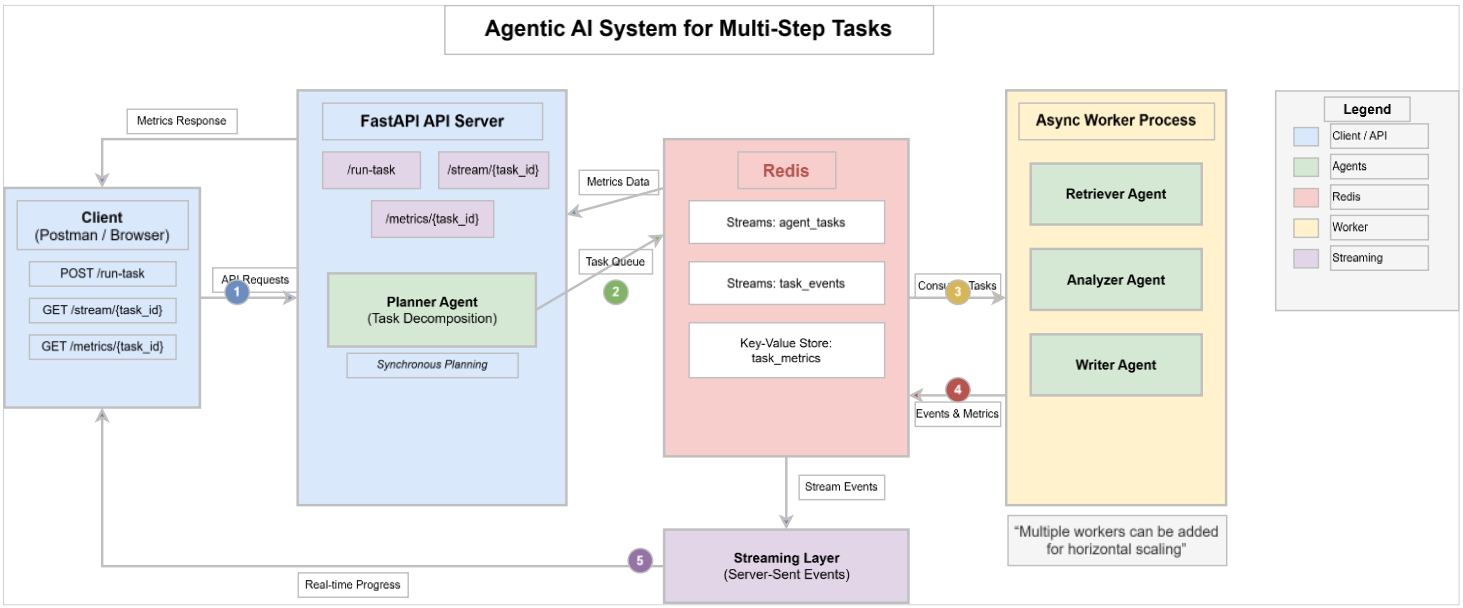


Figure 1 High-Level Architecture of the Agentic AI System

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.

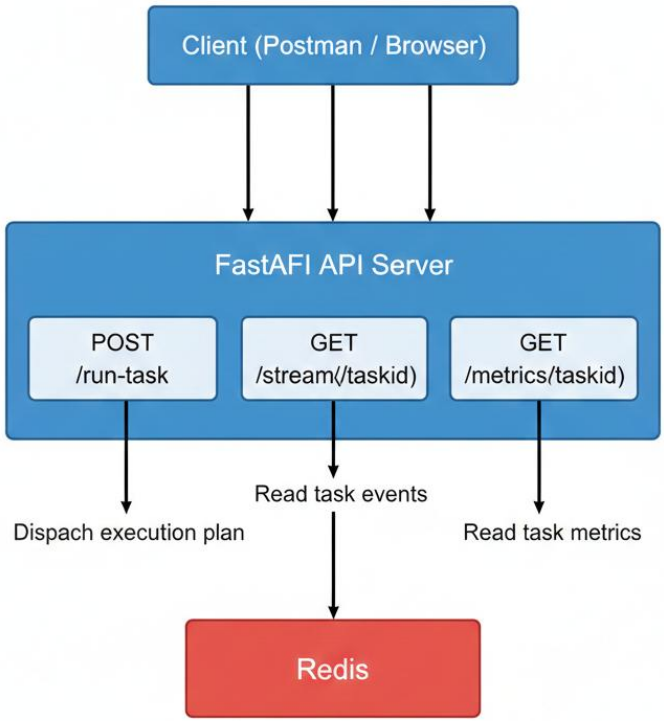
The diagram visually highlights Redis as the central coordination backbone, the separation between API orchestration and worker execution, and the real-time streaming and metrics paths back to the client.

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.

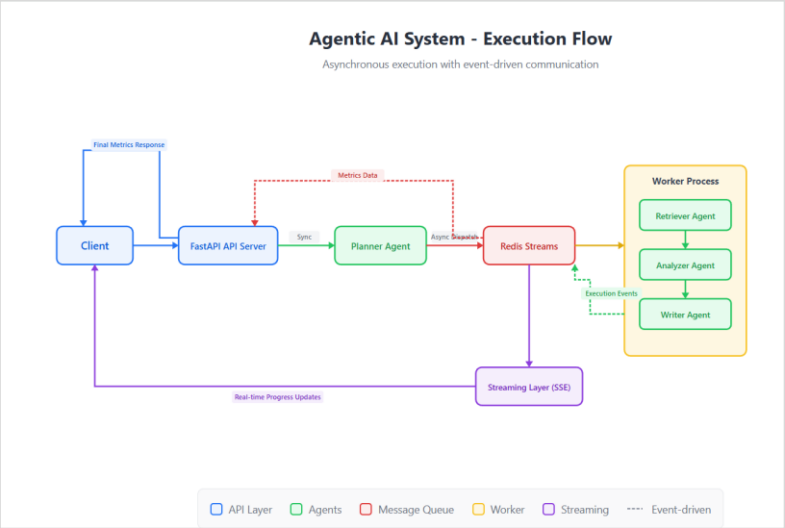
3.5 State Management and Streaming

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 API layer performs only orchestration and streaming. Worker processes run separately and execute all agent logic. There are no synchronous waits between system components.

Redis fully decouples the execution lifecycle, enabling horizontal scaling and fault isolation.

6. Manual Batching Strategy

Steps targeting the same agent are batched together before being pushed to Redis. This reduces message overhead and improves throughput.

The trade-off is that batching is handled explicitly at the transport layer while keeping execution logic simple and readable.

7. Failure Handling and Retry Strategy

Failures are handled at the step level rather than the task level.

The retry strategy includes:

- Exponential backoff
- Maximum retry limits per step
- Idempotent execution logic

This prevents partial task corruption and enables safe reprocessing.

8. Streaming Strategy

The system uses structured event streaming rather than raw text output. Server-Sent Events were chosen due to their simplicity, native browser support, and low overhead for one-way streaming.

9. Scalability Considerations

The system scales horizontally by adding more worker processes. API servers remain stateless and Redis handles coordination.

Potential future improvements include per-agent streams, persistent databases for state, and distributed worker deployment across nodes.

10. Trade-offs Made

Key trade-offs include choosing Redis Streams over Kafka for simplicity, using in-memory logic instead of a database for state, selecting SSE over WebSockets, and avoiding black-box agent frameworks in favor of explicit orchestration.

These trade-offs were intentional and documented.

11. Post-Mortem Analysis

A key scaling issue encountered was that in-memory metrics were not visible across processes, which required migrating metrics storage to Redis.

A design decision that would be changed in the future is persisting task state externally instead of relying on transient in-memory tracking.

Key lessons learned include the importance of explicit state ownership, observability, and careful orchestration in agentic systems.

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.