System Design Document

Agentic AI System for Multi-Step Tasks

---

1. Overview

This project implements an **Agentic AI System** capable of handling complex, multi-step user tasks by coordinating multiple specialized agents. Instead of solving tasks monolithically, the system decomposes a task into logical steps and assigns each step to a dedicated agent responsible for a specific function.

The system demonstrates:

- Agent-based architecture
- Asynchronous orchestration
- Message-queue–driven communication
- Streaming partial responses
- Failure handling and scalability thinking

The design intentionally avoids black-box agent frameworks to expose internal orchestration, retries, and message flows — aligning with real-world production AI system design.

---

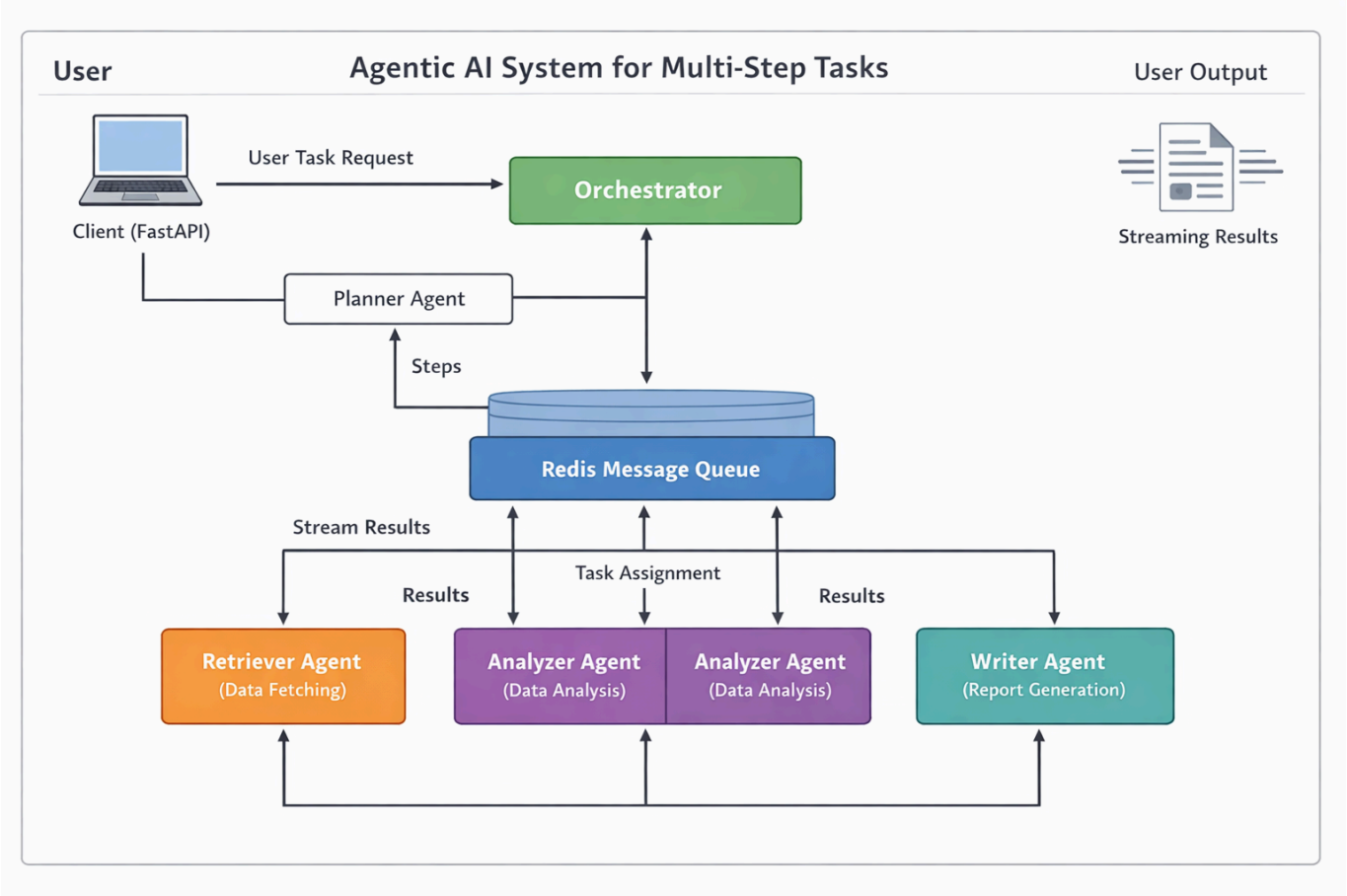
2. High-Level Architecture

Core Components

- **FastAPI Service** – Entry point for user requests and streaming responses
- **Orchestrator** – Coordinates agents and task execution
- **Planner Agent** – Breaks user tasks into steps
- **Retriever Agent** – Gathers contextual data
- **Analyzer Agent** – Performs reasoning and analysis
- **Writer Agent** – Produces the final output
- **Redis** – Acts as the message queue and lightweight state store
- **LLM Layer (Gemini API / Mock Mode)** – Generates agent outputs

---

3. Architecture Diagram



---

4. Agent Boundaries & Responsibilities

4.1 Planner Agent

Responsibility

- Converts a complex user request into an ordered sequence of steps
- Assigns each step to a specialized agent

Why it exists

- Separates planning from execution
- Enables flexible workflows and dynamic task decomposition

4.2 Retriever Agent

Responsibility

- Gathers factual or contextual information

- Acts as a research layer

Why it exists

- Prevents reasoning agents from hallucinating
- Mirrors real-world data ingestion pipelines

4.3 Analyzer Agent

Responsibility

- Performs reasoning, pattern detection, and synthesis
- Operates on retrieved or intermediate context

Why it exists

- Decouples analysis from writing
- Improves clarity and debuggability of reasoning steps

4.4 Writer Agent

Responsibility

- Produces the final, user-facing output
- Focuses on clarity, formatting, and completeness

Why it exists

- Clean separation between reasoning and presentation
- Makes output quality tunable without affecting logic

4.5 Orchestrator

Responsibility

- Central coordination of task flow
- Dispatches steps to agents
- Tracks task state and progress
- Handles failures and completion

Why it exists

- Avoids agent-to-agent tight coupling
- Enables retries, batching, and observability

---

5. Async Orchestration Flow

1. User submits a task via FastAPI
2. Orchestrator assigns a unique `task_id`
3. Task is pushed to `queue_planner`
4. Planner generates a step plan
5. Orchestrator sequentially dispatches steps to agents
6. Each agent processes asynchronously and pushes results to `queue_results`
7. Orchestrator aggregates results and advances the workflow
8. Final result is stored in Redis
9. Client receives streamed updates via SSE

This async pipeline allows:

- Non-blocking execution
- Concurrent agent scaling
- Fault isolation

---

6. Failure Handling Strategy

- Each agent can explicitly return `status: failed`
- Orchestrator halts execution on failure
- Error details are propagated to the client
- Partial results are preserved for debugging

This mirrors production systems where graceful degradation and visibility are critical.

---

7. Mock Mode vs Real LLM Mode

Due to LLM quota limitations during development, the system supports **Mock Mode**.

Mock Mode

- Enabled via `.env`
- Agents return deterministic responses
- Allows full end-to-end testing without API calls

Real Mode

- Uses Gemini API

- Activated by setting `MOCK_MODE=false`
- No code changes required

This design ensures:

- Testability
- Cost control
- Production readiness