Multi-Agent Task Orchestration System

Design Document — Architecture, Trade-offs & Decisions

System Overview: A full-stack platform where four specialized AI agents (Planner, Researcher, Writer, Reviewer) collaborate to produce a comprehensive report from a user prompt. Built with FastAPI (Python) + Next.js (React), using Groq LLaMA 3.1 as the LLM backend and Server-Sent Events for real-time progress streaming.

1. Architectural Decisions

Agent Abstraction — Blackboard Pattern

Each agent inherits from an abstract **Agent** base class with a single **run(task)** → **AgentResult** interface. The entire Task object is passed by reference through all agents (blackboard pattern), so each agent reads prior work and mutates specific fields: **plan**, **research**, **draft**, **feedback**, **final_report**. This keeps agents loosely coupled — the Writer doesn't need to know how the Researcher formatted output, only that **task.research** is populated.

Orchestrator as a Sequential Pipeline with Review Loop

The Orchestrator is a plain async class that sequences agents: Planner → Researcher → Writer → Reviewer, with one revision cycle if the Reviewer returns NEEDS_REVISION. A cap of 1 revision prevents infinite loops while demonstrating the feedback mechanic end-to-end.

SSE for Real-Time Streaming

Server-Sent Events were chosen for the real-time update channel. Each agent calls **emit()** to append structured events to the task's event log, which the SSE endpoint streams to the browser. The frontend uses the native **EventSource** API — no additional library needed.

In-Memory Task Store

Tasks are stored in a Python dict for simplicity. A UUID is assigned per task submission, enabling concurrent tasks. The store is stateless across restarts — intentionally scoped for this take-home assignment.

2. Trade-offs Considered

Decision	Options Considered	Choice & Reason
Real-time updates	Polling / WebSockets / SSE	SSE — unidirectional, no extra libs, HTTP/1.1 native
Agent execution	Sequential / Parallel	Sequential — simpler orchestration, easier to debug
State management	In-memory / Redis / SQLite	In-memory — minimal deps, sufficient for demo scope
LLM provider	Gemini / OpenAI / Groq	Groq — free tier, fast inference, no billing needed
Revision loops	Fixed 1 / Configurable / Infinite	Fixed 1 — prevents loops, demonstrates the mechanic
Frontend state	REST polling / SSE / WebSocket	SSE + fallback polling for task status bar

3. What I Would Do Differently With More Time

Parallel Researcher — Use asyncio.gather() to run sub-task research concurrently — fan-out/fan-in pattern in the orchestrator. Would significantly reduce total pipeline time.

Persistent Storage — Replace the in-memory dict with SQLite (via SQLModel) or Redis so task history survives server restarts and enables horizontal scaling.

Agent Configurability — Let the user toggle pipeline steps (e.g. skip review, add a Fact Checker agent) via the UI, with a dynamic orchestrator that reads a pipeline config.

Token Streaming — Stream LLM tokens word-by-word into the UI for a more alive real-time feel, rather than waiting for the full response.

Error Recovery — Per-agent retry logic with exponential backoff; surface partial results if one agent fails mid-pipeline so work isn't lost.

Unit Tests — Test Orchestrator state transitions and the revision loop with mocked agents. Component tests for the pipeline visualizer in the frontend.

4. Assumptions Made

• Real LLM responses are more valuable than hardcoded stubs — Groq provides free, high-quality inference.

• A single revision cycle satisfies the 'reviewer can send it back' requirement without needing a configurable loop count.

• CORS is open to localhost only — single-developer local setup, no auth required for take-home scope.

• The four-agent pipeline (Planner → Researcher → Writer → Reviewer) is the canonical sequence; no parallel branches needed for core requirements.

• In-memory task store is acceptable since the assignment does not mention task history or persistence as a core requirement.

5. Technology Stack

Layer	Technology	Purpose
Backend Framework	FastAPI (Python)	Async API with SSE support
LLM Provider	Groq — LLaMA 3.1 8B	Free, fast inference for all 4 agents
Data Validation	Pydantic v2	Typed models for Task, AgentResult
Frontend	Next.js 14 (App Router)	React with TypeScript
Styling	CSS Modules	Scoped, no framework overhead
Real-time	Server-Sent Events	Native browser EventSource API
State (backend)	Python dict (in-memory)	Simple, sufficient for demo scope

THE END

G.AJAY KUMAR
ajaykumargandarapu@gmail.com