Aura-Flow: Internal Architecture & Documentation

1. Executive Summary

Aura-Flow is an advanced, multi-agent artificial intelligence orchestrator capable of planning, coding, and self-testing logic sequences dynamically. It mitigates the hallucination risks typical of standard Large Language Models (LLMs) by delegating responsibilities into a cyclic, self-healing loop utilizing LangGraph state machines.

The project addresses the growing demand for **Autonomous Software Engineering**, allowing non-technical or semi-technical users to generate highly specific algorithms simply by stating task directives, all monitored through an elegant, production-grade interface.

2. Core Problem Addressed

Standard LLM chat sessions have limitations:

1. They often produce hallucinated or syntactically incorrect code.
2. They are static—if an error occurs, the user must manually copy paste the error back to the model.
3. They lack multi-step determinism. One massive prompt yields low-quality architectural decisions compared to specialized agents routing information sequentially.

Aura-Flow Solves This By: Creating a secure node loop. A **Researcher** builds the mental model. A **Coder** writes the file. A **Verifier** runs `node <script>` natively in a sandbox. If the execution fails, the verifier extracts the `stderr` string and automatically reroutes the graph back to the **Coder** along with the error log. The system loops until the tests report zero errors.

3. Architecture & State Management

Aura-Flow uses a bidirectional communication architecture separating heavy ML execution states in Python from lightweight event-rendering in React.

A. Backend - Orchestrator Engine (FastAPI & LangGraph)

- **Agent Node Graph**: Built using `langgraph`. The state machine has the following structure:
  - `START` → `Supervisor`
  - `Supervisor` evaluates intention. If valid task → `Researcher`.
  - `Researcher` → Generates internal notes and code logic.
  - `Coder` → Extracts notes, parses JSON, and writes raw code via a `write_file` mock sandbox tool.
  - `Verifier` → Uses the Python `subprocess` module to execute the code generated using Node or bash.
  - Iteration Check: If `Verifier` detects an exit code `> 0`, it updates the global graph state with the errors and returns control to `Coder`.
  - Success Check: If `exitcode == 0`, transition to `END`.
- **Large Language Model**: Originally built for OpenAI context parsing, Aura-Flow was refactored to utilize **Google Gemini 2.5 Flash** due to latency efficiency and a generous free tier. The standard `.with_structured_output` wrapper was stripped and custom template sanitization was integrated to avoid Langchain `f-string` injection vulnerabilities when passing raw JavaScript curly braces `{}`.

B. Frontend - Dashboard (Next.js & React 19)

- **Rendering Scheme**: Utilizes Next.js App Router entirely composed in a `'use client'` environment for real-time reactivity.
- **Server-Sent Events (SSE)**: Standard WebSockets were overkill for streaming textual thought-processes. Instead, the frontend fetches the `/api/run-flow` endpoint and hooks into `TextDecoder` streams to push real-time agent thoughts into the UI log as they happen without awaiting massive JSON resolutions.
- **Aesthetic**: **Glassmorphism**. Deep backdrop blurs (`backdrop-filter`) and calculated border-opacity adjustments present a fluid feel. The dynamic layout intelligently manages responsive CSS variables scaling smoothly from Dark Mode Obsidian (`#050505`) to Light Mode slates (`#f8fafc`).

4. Current State & Future Extensibility

Currently, the Verifier runs raw bash commands utilizing Node.js installed natively on the host operational system. For production enterprise environments, the `verifier_node` execution context should be swapped with isolated Docker containers (e.g. `docker run --rm -v $(pwd):/app node:alpine node /app/sandbox.js`) to prevent arbitrary code execution vulnerabilities on the host machine.

Future Hooks:

1. Integrating specialized nodes for `FrontendEngineer` capable of utilizing Vite or React compiler sub-routines.
2. Building a local persistence database for past conversation thread retrieval.