Architecture Document

At a high level, the system is a **client-server platform**: a web UI lets the user submit a problem (vague text) or user code. The **backend** is a FastAPI service with key endpoints: `/api/generate-problem` (orchestrates Stages 3–5 and streams progress) and `/api/evaluate` (handles Stage 6 user submission). There is also `/api/improve/problem/{id}` for user challenges (Gatekeeper and healing).

The **backend components** include:

- **LLM Inference:** All natural-language processing is done via a fine-tuned GPT-4.1 model (accessed through OpenAI's API). The code uses `client.chat.completions.create(model=FINE_TUNED_MODEL, ...)` for each stage. The model ID (`gpt-4.1-mini-2025-04-14` fine-tuned) and prompt templates are stored in code.

- **Orchestrator:** Custom Python functions (`stage3_generate_specification`, `stage4_generate_tests`, etc.) implement each stage's logic. FastAPI streams status updates (JSON-ndjson) to the client during long operations. These functions call the LLM, run local validations, manage retries, and aggregate results.

- **Database (Postgres):** SQLAlchemy ORM defines tables to persist data:

    - **Problems**: stores `problem_id`, title, description, constraints, examples, tags, starter code, evaluation type, and the LLM prompt used.

    - **TestCases**: for each problem, stores a JSON list of input tuples and expected output for each generated test.

    - **ReferenceSolutions**: the final optimized solution code per problem (unique).

    - **Submissions**: each user submission entry with code, passed/test counts, timestamp.

    - **SubmissionResults**: detailed per-test results (input, expected, actual, pass/fail) for each submission.

- These tables are created at startup (`Base.metadata.create_all`) and read/write are done within the FastAPI endpoints. The database enables caching of generated problems and user history.

- **API Layers:** FastAPI exposes endpoints with Pydantic models. For example, `GenerateProblemRequest` has a `vague_problem: str` field, and the response includes `problem_id`, `title`, `starter_code`, and `test_case_count`. User submission evaluation expects `{problem_id, user_code, [user_id]}` and returns a structured `EvaluateResponse` (with passed/total/percentage). CORS middleware allows any origin for demo purposes.

- **LLM Prompt Orchestration:** Each stage has a carefully crafted system prompt. Stage 3’s prompt defines JSON schema (title, tags, etc.) and strict rules (function signature, constraint logic). Stage 4’s prompt (see below) instructs the model to output two Python functions (`generate_inputs` & `is_valid_input`) covering five test categories. Stage 5 uses a system prompt for writing code under “OPTIMIZED” or “BRUTE FORCE” modes. A separate “healing” prompt is issued if mismatches occur (see Stage 5 below).

- **Self-Healing & Circular Healing:** The orchestrator embeds retry loops. Stage 4 and Stage 5 include “circular healing”: if the reference solution rejects too many tests (TestQualityFailure) or optimized vs brute outputs disagree, we loop back to regenerate tests or re-run solution generation. These mechanisms are implemented by catching custom exceptions and feeding the feedback into new LLM calls. In architecture, this means the pipeline can repeat Stage 4 and/or Stage 5 multiple times before finalizing.

- **Deployment:** The code runs as a Uvicorn-based FastAPI server on port 8000. It integrates the OpenAI API and connects to Postgres (configured via `DATABASE_URL`). The `FINE_TUNED_MODEL` constant is set to the fine-tuned GPT ID. CORS and JSON streaming are enabled for real-time client updates.