Design Document

System Objectives: Build an AI-assisted coding practice platform that generates a complete problem-solving environment (formal specification, test suite, reference solutions, and an IDE) from minimal or vague user input. We understand a candidate after giving an interview “n” days ago may not be able to recall the problem fully well, so they can give a vague statement to our platform, and we will do our best to generate a coding environment for it. Whether a user pastes a known LeetCode statement (also called a formal specification) or recalls a problem in their own words, the system formalizes it, creates diverse tests, and provides a reference solution for practice. The platform emphasizes **robustness** and **trust**: any errors (in the spec, tests, or solution) can be flagged by the user with counterexamples, triggering automated self-healing to correct mistakes.

User Personas: We model four input “personas” reflecting how different people describe problems after an interview:

- *Layman:* Simple, non-technical phrasing. “Interview recall without terminology”. (e.g. “find the common string”).
- *Conversational:* Normal speech with some detail. “Human-to-human explanation”.
- *Technical Shorthand:* Shorter, jargon-heavy descriptions. “Abbreviated interview notes”.
- *Implementation-Specific:* Emphasis on code/pseudocode hints.

These personas simulate real users who might recall a problem vaguely. By fine-tuning on examples of these four styles, the platform can parse any user query without requiring formal math language.

Design Principles: We follow **completeness**, **correctness**, and **self-healing**. The system is **user-centric** (handles incomplete input), **reliable** (multiple verification layers), and **iterative** (allowing feedback loops). Key practices include:

- **Multi-stage pipeline:** Each problem goes through distinct stages (specification, test generation, solution generation, evaluation) to modularize complexity.
    - Stage 1: Formal Specification to Vague Statement (Training)
    - Stage 2: Fine-tuning vague and formal specification (Training)
    - Stage 3: Vague Problem Statement to Formal Specification (Inference)
    - Stage 4: Test Suite Generation (Inference)
    - Stage 5: Reference Solution Generation (Inference)
    - Stage 6: Candidate Solution Validation.
    - Stage 7: Candidate Feedback with Counterexamples.
- **Formal specifications:** Converting to a precise spec ensures clarity and correctness, following strict formatting rules (e.g. function signature

`solve(self, …)`, sample examples and constraint extraction).

- **Automated test "oracle":** LLM-generated test cases are validated by a local `is_valid_input` function (oracle) to ensure they respect constraints.

- **Brute-force baseline:** Brute-force solutions serve as ground truth for correctness; optimized solutions are compared against them (healing) to detect and fix errors.

- **Self-healing/cross-healing:** If tests or solutions fail (either fails Oracle or conflict between brute/optimized), the system automatically retries with adjusted prompts or logic until consistency is achieved (circuit-breaker strategy using max retries)

- **User feedback loop:** End users can provide counterexamples; a "Gatekeeper" audits them against constraints and, if valid, injects these into the pipeline to regenerate tests/solutions (truth-injection healing).

- **Scalability & efficiency:** The design assumes large DSA inputs; for stress tests it uses sparse representations (e.g. Python repeat syntax) and timeouts to prevent resource overuse.

Fine-Tuning Rationale: We curated **215 LeetCode problems** (various categories and difficulties) from a public dataset. For each, we took the formal LeetCode statement and generated four "vague" variants (one per persona) in English. This produced (informal description → formal specification) training pairs. We then fine-tuned a GPT-4.1-mini model on this dataset using supervised fine-tuning (SFT). The result is a specialized LLM that can translate user queries in any of the four styles into a precise problem spec (JSON with title, description, constraints, starter code, etc.). By anchoring on real LeetCode problems and these personas, the model learns to handle ambiguous or paraphrased inputs robustly.

Training & Translation Strategy: The HuggingFace LeetCode dataset provided ground-truth problem specs. For each, we wrote four persona-flavored prompts (e.g. an interviewee's layman description) as input, and the original spec as output. During inference, the fine-tuned model uses a strict prompt (e.g. "Convert this vague question into a formal spec") to generate JSON output. This supervised approach ensures alignment between informal recollection and formal problem definition.