Implementation Document - Parameters consideration.

Prompt Design and Temperature Control

Each pipeline stage uses a dedicated system prompt with carefully chosen temperature settings to balance determinism and diversity.

- **Stage 3 (Vague Input → Formal Specification)**
  - Temperature: **0.0**
  - Objective: deterministic generation of a strict JSON specification.
  - The prompt enforces:
    - Required fields (title, description, constraints, examples, starter code).
    - A fixed function signature (`class Solution: def solve(self, …)`).
    - Python-style constraint expressions.
- **Stage 4 (Test Case Generation – Oracle)**
  - Initial temperature: **0.5**, increased by **+0.2 per retry** (up to 0.9).
  - The model generates exactly two Python functions:
    - `generate_inputs()` — produces categorized test inputs.
    - `is_valid_input(*args)` — validates inputs against constraints.
  - Feedback from prior failures (invalid tests, low yield, downstream mismatches) is injected into subsequent retries to encourage correction and diversity.
- **Stage 5 (Reference Solution Generation)**
  - Optimized solution generation: **temperature = 0.2**.
  - Brute-force solution generation: **temperature = 0.2**, with instructions favoring correctness over efficiency.

    - Healing iterations (on mismatches): **temperature = 0.1**, minimizing variance while fixing logic.

This staged temperature control ensures deterministic specification, diverse test coverage, and controlled correction during healing.

Retry Logic and Circuit Breakers

The system includes explicit retry and termination policies to prevent infinite loops.

Stage 4 – Oracle Retries

- A minimum of **10 valid test cases** must be produced after filtering.
- Failures that trigger regeneration:
    - Missing required functions.
    - Forbidden AST patterns.
    - Runtime errors in test-generation code.
    - Fewer than the minimum valid tests.
- Maximum retries: **3**.
- Each retry includes diagnostic feedback from prior attempts.

Stage 5 – Solution Healing

- The optimized solution is compared against a brute-force solution (or user-provided truth).
- On any mismatch:
    - A healing prompt is constructed with:
        - The failing input.
        - Optimized output.
        - Expected output (from brute-force or user).

- Maximum healing attempts: **3**.
- If healing fails or code no longer compiles, the last viable optimized solution is retained.

Circular Healing

- If a generated solution invalidates more than ~30% of the test suite, the system treats this as a **test quality failure** and regenerates the test suite (Stage 4), preventing brittle or adversarial tests.

Timeout-Safe Execution Model

All LLM-generated code is treated as untrusted and executed in **isolated child processes** with hard time limits.

- **Execution model**
  - Code compilation and execution occur *inside* child processes (not the parent).
  - Communication uses multiprocessing queues to safely capture outputs or errors.
- **Timeouts**
  - Stage 4 (test generation): **6 seconds**
  - Stage 5:
    - Optimized solution: **2 seconds**
    - Brute-force solution: **4 seconds**
  - Stage 6 (user submissions): **2 seconds per test case**
- On timeout:
  - The process is terminated.
  - A `"TIMEOUT"` result is returned and handled gracefully.

This design prevents infinite loops, runaway recursion, and resource exhaustion.

AST-Based Static Validation

Before any generated Python code is executed, it undergoes static analysis using Python's `ast` module.

- **Disallowed constructs**
    - `import` and `from … import …`
    - Any filesystem, network, or OS access
- **Structural enforcement**
    - Stage 4 code must define both `generate_inputs` and `is_valid_input`.
    - Stage 6 user submissions must define a `Solution` class with a `solve` method.
- Violations trigger immediate rejection and, where applicable, regeneration.

AST validation acts as both a **security barrier** and a **format correctness check**.

Test Generation and Validation Constraints

The Stage 4 oracle enforces strict compliance with the formal specification:

- Test inputs are filtered through `is_valid_input` to ensure constraint satisfaction.
- Duplicate tests are removed using JSON-serialization-based hashing.
- Tests are grouped into categories (basic, edge, stress, adversarial, domain-specific).
- Large inputs are represented compactly (e.g., `[0] * 100000`) to reduce token usage.

If too many generated tests are invalid or rejected, the oracle is regenerated.

Brute-Force as a Differential Oracle (Not Formal Verification)

The system generates both a brute-force and an optimized solution using LLMs.

- The brute-force solution is treated as a **relative correctness oracle**, not a formally verified reference.
- Correctness is established via **differential testing**:
  - Optimized and brute-force outputs are compared on shared test cases.
- For stress tests, brute-force execution may be skipped to avoid timeouts.
- **User-provided expected outputs override brute-force results** and are treated as absolute truth.

This approach provides practical correctness validation without claiming formal proof.

User Submission Evaluation (Stage 6)

When a user submits code:

1. The code is statically validated (AST check).
2. Each test case is executed in isolation with a timeout.
3. Outputs are compared using the specified evaluation mode:
   - Exact match
   - Order-insensitive match
   - Floating-point tolerance
4. Per-test results and summary statistics are returned and stored.

User Feedback and Truth Injection

Users may submit counterexamples if they believe the system is incorrect.

- A **Gatekeeper** validates user inputs against original constraints.
- Valid challenges trigger:
    - Test-suite regeneration with user inputs injected.
    - Solution regeneration using user outputs as ground truth.
- The database is updated with the healed test suite and reference solution.

This mechanism allows controlled human intervention without undermining system integrity.

Summary of Guarantees and Limitations

- The system **does not claim formal verification**.
- Correctness is established through:
    - Constraint-validated test generation.
    - Differential testing between independently generated solutions.
    - Human truth injection when needed.
- All execution is sandboxed and timeout-limited.
- Failure modes are explicitly detected and handled.