Workflow/Inference Pipeline Document

The **inference pipeline** has four main stages, plus a feedback loop for user challenges:

1. **Stage 3 – Vague Input → Formal Specification:** The user's free-form text is sent to the fine-tuned LLM (with a system/user prompt) which outputs a JSON problem specification. This spec includes *title*, *problem_description*, *examples*, *constraints* (list of Pythonized constraints), *constraint_source*, *starter_code* (class `Solution` with `solve(self, …)` signature) and *evaluation_type*. For example, the system prompt enforces rules on constraints and output format. The returned JSON is parsed into a `spec` dictionary. (If the output is fenced code, the code strips markdown.) We then store the core fields in the database.

2. **Stage 4 – Test Case Generation (Oracle):** Next, the platform calls the LLM with a "Senior QA Engineer" prompt to produce two Python functions: `generate_inputs()` (returns a dict with five categories of test-tuples) and `is_valid_input(*args)` (returns `(bool, reason)` based on the constraints). The system prompt instructs the model to: analyze constraints, plan edge cases, obey token limits, etc., and to output **exactly** those two functions. After receiving the code, we statically validate it (no forbidden `import`, required functions present), then execute it safely (in a subprocess with a 6s timeout). We call `generate_inputs()` and filter all returned inputs through `is_valid_input()`, discarding invalid or duplicate cases.

   The goal is to produce **≥10 valid test cases across all categories**. If fewer than 10 valid inputs are obtained or other validation failures occur, an `OracleFailure` is raised and the code retries: the assistant's last answer is fed back with a message like "Your code failed Oracle validation. [Diagnostic info]. Fix it.". Each retry increases randomness (temperature from 0.5 up) to encourage diversity. Typically up to 3 attempts are made. The final output of Stage 4 is a *golden* list of input tuples and the categorized dictionary of tests.

3. **Stage 5 – Brute-force vs Optimized Solutions:** The LLM is asked to write two solutions under different modes: an optimized solution and a brute-force solution (using recursion/itertools for correctness). Both receive the problem title, description, constraints, starter code, and examples. They must return only a Python `Solution` class. After generating the code, we compile/check both for syntax errors. We then run **consensus verification**: for each test input (except "Stress" or "Boundary" tests where brute may time out), we execute the optimized solution (2s timeout) and the brute solution (4s timeout). We compare the outputs (with user-provided "ground truth" taking precedence if it exists).

   If the optimized solution output ever mismatches the expected (brute or user) output, we enter a *circuit-breaker* healing loop. A special "enhanced healing" prompt is constructed that describes the failing input, optimized output, and expected output

(source from brute or user). We then ask the model to “fix” the optimized solution (low temperature 0.1) so it matches the truth. This is repeated up to 3 times (MAX_HEALS=3). If even after healing the code fails to compile or still mismatches, we fall back to the last optimized version. Once all tests pass (or brute/optimized agree on all), consensus is reached. The (healed) optimized code becomes the final *reference solution*.

4. **Stage 6 – User Submission Evaluation:** The user is presented with the problem spec, starter code, and test suite. When the user submits their code, we run it against all stored test cases. We spawn a sandboxed subprocess for each test with a 2s timeout. Inputs are converted back into data structures (`ListNode`, `TreeNode`, etc.) and the `solve` method is invoked. We compare each actual output to the golden expected output using the specified evaluation mode (`exact_match`, `list_any_order`, or float tolerance). We collect per-test results (passed/failed, errors) and compute an overall score. The API returns a summary (`passed`, `total`, percentage) and stores the submission/results in the database for record.

User Feedback & Healing Loop: If the user finds an issue, they can submit a set of counter-examples (challenge inputs and their expected outputs). The system invokes the *Gatekeeper* (`analyze_user_challenge`) to verify these against the original constraints. If any input violates hard constraints, we reject the challenge with a reason. If the inputs are valid (decision “YES”), we initiate **system healing**:

- *Stage 4 (Retest):* We re-run test generation but include the user’s cases as a new category “User Provided Edge Cases”. We ensure these inputs are properly formatted (wrapping as needed) and inject them into the test suite.

- *Truth Injection:* The user’s expected outputs are treated as absolute truth. The Stage 5 consensus loop then uses those values instead of brute-force for those inputs (hard override).

- *Stage 5 (Resolution):* We regenerate the optimized solution with the updated test set and user truths.

- *Database Update:* The reference solution and test cases in the database are replaced with these healed versions.

Finally, we return a status “HEALED” to the user, showing the updated test count. This loop ensures that genuine oversights in the AI-generated tests or code can be corrected using human guidance.