💫 Perfect Assertions

Project Introduction

🌻 A Message For a Perfect Assertion Dev! 🌻

Hey! Congratulations on being invited to this project! This means you're an engineer with the best algorithm-writing skills out there!

Your main goal is to provide the reference solution (the golden code response) that passes a set of valid and comprehensive unit tests (golden unit tests).

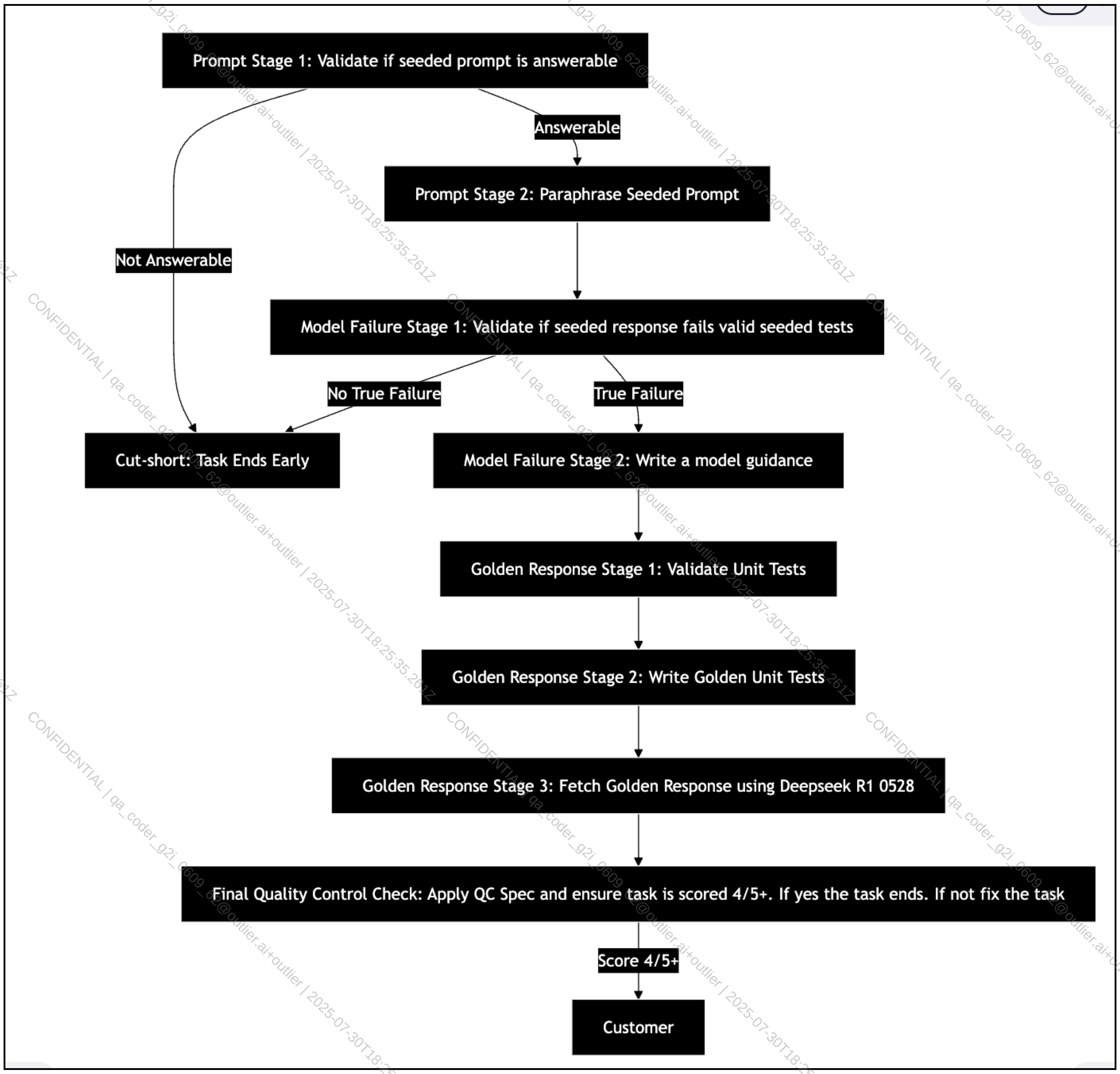
Together, we'll create a dataset designed to help AI models improve in the Aider Benchmark, where AI models are asked to rewrite the code into the reference solution for a problem.

You'll perform a similar, exciting task in this exciting project that has just kicked off. Let's make it successful together in the coming weeks.

🌹

Attempter Workflow Diagram

The diagram below gives you a brief overview of the attempter workflow


Prompt Stage 1: Validate if seeded prompt is answerable
Answerable
Not Answerable
Prompt Stage 2: Paraphrase Seeded Prompt
Model Failure Stage 1: Validate if seeded response fails valid seeded tests
No True Failure
True Failure
Cut-short: Task Ends Early
Model Failure Stage 2: Write a model guidance
Golden Response Stage 1: Validate Unit Tests
Golden Response Stage 2: Write Golden Unit Tests
Golden Response Stage 3: Fetch Golden Response using Deepseek R1 0528
Final Quality Control Check: Apply QC Spec and ensure task is scored 4/5+. If yes the task ends. If not fix the task
Score 4/5+
Customer

🚨 Using AI Tools

Directly copying and submitting content from AI tools like ChatGPT without significant modification is prohibited.

While you're encouraged to use AI for critique and suggestions, you must personally implement and refine the work.

You'll be asked to use an LLM to write the starter golden response code. **Significantly rewrite this code.**

🌟 Success Criteria for Stellar Task

Follow these steps with precision, and you will **always** produce a Stellar Task (tasks scored highly).

⭐ **North Star**	Provide the most optimal reference solution that passes a set of comprehensive and correct unit tests by rewriting the skeleton code
🚨 **Baseline**	Develop a very comprehensive test suite first, and ensure you test all crucial/core aspects of the algorithm with distinct, meaningful tests, as this will make validating your golden code response straightforward.
True Failure	We want problems that make state-of-the-art AI models fail on correct/valid unit tests
Important documents	🚨 Please ensure you have the following documents open as you task: - [WIP Perfect Assertions] Project Specifications - Project Instructions - [Perfect Assertions] QC Spec Doc 4 Attempters - Quality Guidelines
Clarifying Question	*Does my test suite cover all aspects of the code, and would passing the test suite mean my reference solution optimally solves the problem to the fullest?* 🏆***If YES -> I have a golden code response for a golden test suite***🏆 ❌ ***If NO ->I have to circle back and fix the unit tests & reference solution*** ❌

Important Project Metrics

Metric	Value
# Turns	1
Total Claim Time	2.5h
Max Paid Time	1.5h
Recommend Task Pausing	10-15m brake every 1h is recommended

Working Environments

You'll be shown two working environments (IDE called 'Sphere Engine').

#	Name	Can be modified	Description
1	**[First Workspace] Failure Confirmation**	No - Read Only	The first workspace appears after the prompt stage. It's read-only and contains a preloaded code response (the seed response) and a provided unit test suite (seed unit test). The purpose of the first working environment is to confirm that the seeded code response fails a valid unit test. You'll be looking for a fail-to-pass unit test.
2	**[Second workspace] Golden code response & Unit Test Development**	Yes - Editable	The second workspace appears after evaluating the provided unit tests. Here you'll implement the provided skeleton code functions, develop a comprehensive test suite that meets all criteria, and ensure your final code passes those tests.

Type of Unit tests

You'll be working with three different types of unit tests depending on the task stage.

Test Type	Validity	Action
Pass to Pass ✅ *A valid unit test-used to validate a **golden response/reference solution***	**VALID ✓**	**Keep in/Add** to the golden test suite
Fail to Pass ✅ *A valid unit test—used to validate a **model failure***	**VALID ✓**	**Keep in/Add** to the golden test suite
{Pass/Fail} to Fail ❌ *A faulty unit test –with an incorrect expectation (expects/asserts a wrong value)* *The unit tests need to be **removed** before testing the code*	**INVALID ✗**	⚠️**REMOVE** from test suite ⚠️

Five Golden Unit Test Criteria

#	Criteria	Description
1	**High Value Code Coverage**	Do the tests cover core logic and critical paths, not just boilerplate?
2	**Meaningful Assertions**	Does each test verify specific, important behavior, not just that the code runs?
3	**Edge Case & Input Variety**	Does the unit test suite include tests for nulls, empty data, min/max values, invalid inputs, etc?
4	**Test Independence**	Do the tests not rely on shared state or order of execution?
5	**No Flaky or Redundant Tests**	Do the tests always pass/fail deterministically? Are there any duplicates doing the same check?

Project Terminology

#	Concepts	Description
1	**Golden Code Response**	Complete, correct, and efficient code that fully satisfies the prompt and passes all unit tests.
2	**Golden Unit Test Suite**	A comprehensive test suite fulfilling all 5 unit test criteria. It's used to validate the golden response by testing the main functions already present in the skeleton code (skeleton code contains the main functions). The Golden unit test suite is located inside the 'test' file, whose extension may vary depending on the coding language of your task. You can find that file in the 2nd working environment, and the content should be provided by you.
3	**Skeleton Code**	Basic code framework with main function signatures to resolve the prompt. You implement the main functions and add helper functions as needed, but only test the main functions in your test suite!
4	**Skeleton Code File Name**	Pre-determined name for the file containing the seeded code response in the 1st working env & the golden code response in the 2nd working env.
5	**Seeded Prompt**	The original prompt was provided at task start. Use to fetch the seeded code response.
6	**Paraphrased Prompt**	Rewritten version (distinct semantics) of the seeded prompt with different wording but same (pragmatic) meaning and length.
7	**Seeded Code Response**	Initial code response to evaluate for failures against unit tests and which it's located in the 1st working environment.. The name of the file depends on the variable [skeleton_code_file_name]+.[file_extension].
8	**Seeded Unit Tests**	A set of unit tests initially used for failure validation, and afterward, modified into a golden unit test suite (editable 2nd env). The seeded unit tests are located in the test.[file_extension] file in the 1st read-only working env.
9	**True Model Failure**	The seeded code response fails a valid unit test present in the seeded unit tests before changes.
10	**Valid Unit Test**	Unit test that checks core functionality with correct expected values.

To Conclude

Your task is to provide the **correct code implementation** by first **paraphrasing the prompt** and **confirming a true model failure.** Then, you'll use the provided **skeleton code** and **unit tests** to develop a **comprehensive unit test suite** that is used to validate that your final code is a **golden code response..**

Appendix

1. Coding Language Specifications

Language	Test Framework	Test File	Seeded Code Response File	Header File	Additional Information
Python	Pytest / Unittest	test.py	[skeleton_code_file_name].py	N/A	- The test.py file must import at least one of the unit testing libraries, unittest or pytest.
C++	Catch2	test.cpp	[skeleton_code_file_name].cpp	[skeleton_code_file_name].h	- The header.h is used only to define classes, variables, functions, and also to initialize variables. - The same namespace should be used across all the files. -Always include guards to prevent a header from being processed more than once in the same compilation unit, thereby avoiding redefinition errors. You can use traditional #ifndef/#define guards or #pragma once. - The code logic should only be included inside [skeleton_code_file_name].cpp file.

2. CPP files examples

Here is an example of the three different files you may find:

[skeleton_code_file_name] -> circular_buffer

circular_buffer.cpp

```
C/C++
#include "circular_buffer.h"

namespace circular_buffer {

// Your core logic here

}  // namespace circular_buffer
```

circular_buffer.h
(This file is only visible in the Sphere Engine, and not in the taxonomy. feel free to edit it as needed)

```
C/C++
#if !defined(CIRCULAR_BUFFER_H)
#define CIRCULAR_BUFFER_H

namespace circular_buffer {

//Define classes, variables, functions, and also to initialize variables

}  // namespace circular_buffer

#endif // CIRCULAR_BUFFER_H
```

test.cpp

C/C++

```
#include "circular_buffer.h"
#ifdef EXERCISM_TEST_SUITE
#include <catch2/catch.hpp>
#else
#include "test/catch.hpp"
#endif

#include <stdexcept>

// Circular-buffer exercise test case data version 1.2.0

TEST_CASE("reading_empty_buffer_should_fail")
{
    circular_buffer::circular_buffer<int> buffer(1);

    REQUIRE_THROWS_AS(buffer.read(), std::domain_error);
}

#if defined(EXERCISM_RUN_ALL_TESTS)
TEST_CASE("can_read_an_item_just_written")
{
    circular_buffer::circular_buffer<int> buffer(1);

    REQUIRE_NOTHROW(buffer.write(1));

    int expected = 1;
    REQUIRE(expected == buffer.read());
}
...
...
...

//The rest of the unit tests here

#endif  // !EXERCISM_RUN_ALL_TESTS
```