2025 NIST GenAI Code Pilot Challenge Evaluation Plan

NIST GenAI Team

December 4, 2025

Abstract

The NIST Generative AI (GenAI) program is launching the Pilot Code Challenge, a new track that assesses AI systems' ability to verify the correctness of basic Python code. The goal is to advance Large Language Models (LLMs) in writing effective software tests. Participation is open to all who agree to abide by the rules and procedures of the pilot.

Contents

DISCLAIMER

Certain commercial equipment, instruments, software, or materials are identified in this document to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement by NIST, nor necessarily the best available for the purpose. The descriptions and views contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NIST or the U.S. Government.

Revision History

2025-12-01. Version Updated with GenAI Code Pilot Round-2 guidelines.

2025-07-16. First Version.

1 Introduction

This NIST Generative AI (GenAI) program[1] is launching a Pilot Code Challenge, an evaluation that invites participation from academia, industry, and government research organizations. The GenAI program provides a platform to test and measure generative Artificial Intelligence (AI) technology, including AI content generators and detectors. The Pilot Code Challenge focuses on generators, specifically exploring their ability to create code for software testing.

Generative AI is increasingly being used to generate software code, streamlining and optimizing software development. The Pilot Code Challenge aims to assess the capabilities of AI in this area, focusing on a key question: **How well can AI systems generate code to test elementary-level software?** In this context, "elementary software" refers to code with at most two methods each with 30 lines or less of code.

This GenAI Code Pilot will function in two rounds, where the first round is called *Round-1* and the second round is called *Round-2*. This updated version of the evaluation plan includes the instructions for both the first round and the second round of the GenAI Code pilot. Unless stated otherwise, all instructions are the same for both rounds. Updates with the additional instructions are in Section 3.4. Moreover, Round-2 will follow the same protocols and procedures as the initial Round-1 pilot, and **the Round-2 input and output data formats will be the same as used in Round-1**. Although Round-2 will require new submissions as the input file will be different, we anticipate Round-1 systems that submitted can be used without further adjustments to submit on Round-2. All are welcome to participate in Round-2. Registration or participation in the Round-1 pilot are not prerequisites for signing up for and participating in the Round-2 pilot.

2 Task Overview

The Pilot Code Challenge focuses on *test code generation*. The task is to develop Generative AI models that:

- Take a text specification that describes the software's functionality, and
- Output Python code that tests the correctness of the software.

Test Input: A text *specification* that includes the method header, method names, parameters, and type information (no examples will be provided).

Test Output: Python 3 test code that can be run with the pytest test package, including necessary library imports.

The Pilot Code Challenge consists of a series of independent trials, each with a test input and specification. Details on input and submission formats can be found in Sections 3.4 and 4 respectively.

The challenge will be conducted under two prompt conditions:

- **Fixed prompt:** Participants must use the provided specification as the prompt for their LLM system.

[1] https://ai-challenges.nist.gov/genai

- **Custom prompt:** Participants can create their own text specification for the prompt, with up to 9 custom prompts allowed per trial.

Both fixed and custom prompt conditions are required.

3 Data Overview

3.1 Evaluation Data Set

The Pilot Code Challenge uses a *code bank* generated by NIST, consisting of code specifications with three corresponding Python programs: a correct implementation and two incorrect implementations. The code bank was partly adapted from HumanEval [2] and MBPP (Mostly Basic Programming Problems) [1], with modifications to ensure consistent formatting.

The code bank is categorized into two complexity levels:

- **Simple:** Single method of at most 15 lines of code.
- **Extended:** 1 or 2 methods of at most 30 lines of code in each method.

The code bank contains 40 simple and 10 extended specifications. NIST will test submitted test outputs against the three Python programs associated with each specification:

1. `code_correct`: correct implementation producing the desired output.
2. `code_incorrect_1`: incorrect implementation with mutated error.
3. `code_incorrect_t`: incorrect implementation without TypeError and ValueError checks.

3.2 Development Data Set

Registered participants will receive a small development data set to build, train, and test their system before the NIST Pilot Code Challenge evaluation. This development code bank includes 4 problems (3 simple and 1 extended) that are not in the pilot test set.

The development data set includes:

- Problem file.
- Ground-truth key file, (see Appendix, Section B for details).
- One baseline submission.
- Correct and incorrect code implementations.

3.3 Test Input Data Format

Participants will receive the evaluation trials in a .json file containing a list called `code_list`. Each entry in the test represents a code under test and includes the following fields:

1. `trial_id`: Unique identifier of the trial.
2. `primary_method_name`: Name of the primary (outermost) method.

3. `specification`: Text description of the code's desired function, including the method header with parameter names and order.

4. `prompt_fixed`: Default fixed prompt for fixed prompt condition.

The .json file will format strings with quotes, escaped newlines, and preserved spaces. An example input is provided in Appendix (Section A.2).

3.4 Round-2 Pilot Updates

We updated the data for Round-2 based on the submissions from Round-1. In particular, we revised many of the specifications to handle ambiguous cases, and we removed a few problems from Round-1. In addition to specific changes for each code, we made three general changes, summarized here:

1. **Type Checking**: In Python, `isinstance(x, int)` considers Booleans as integers.

   (a) **Round 1**: Our code was designed to treat Booleans as non-integers.

   (b) **Round 2**: We will use Python's standard type checking behavior, where `isinstance(x, int)` returns True for both integers and Booleans. This means that any boolean will be considered an integer.

2. **Empty List Input**: We have defined the expected behavior for programs expecting non-empty lists of a specific type but receiving an empty list.

   (a) **Round-1**: Programs were allowed to run on empty lists and return results.

   (b) **Round-2**: To ensure consistency, most programs will now raise a ValueError when given an empty list as input. However, all programs with list inputs will have their behaviors defined in their specifications.

3. **Handling Negative Integers:** We have defined the expected behavior for negative integers, as many of the codes were expecting a positive integer as input.

   (a) **Round-2:** Specifications were modified to specify what should happen when integer inputs are negative.

4. **Error Handling**: We acknowledge that timeouts and out-of-memory errors can occur.

   (a) **Round-2**: An entire submission will fail if even one problem has a memory error or causes a time-out.

5. **Submissions:** We increased the submission limit to 10 submissions per team per 24-hour period.

4 Submission Guidelines

Submissions can be made at any time during the submission period, with a limit of 10 submissions per 24-hour period per team. Submissions will be scored within a 30-minute compute time limit; any submission taking longer will be rejected. A scoreboard will display scores for each team and submission, separated by condition (fixed prompt or custom prompt). Submissions on the pilot data

are required. Dry-run submissions on the development data set are optional but will neither appear in the scoreboard nor be analyzed.

4.1 Submission Format

Each submission (test output) should be a single .json file. The file should contain the following metadata fields:

1. `name`: A descriptive name for the submission (e.g., "NIST Baseline Two-Test Submission, Dry-Run Problems")
2. `system`: The name of the system (using only letters and underscores, e.g., "nist_baseline_two_test")
3. `version`: The version number of the submission (matching the input problem json file (e.g., "1.0")

The .json file should also contain a `code_list` list, with one object per code under test. Each object should have the following fields:

1. `trial_id`: The id of the trial
2. `prompt_number`: A number specifying the submission type (0 for fixed prompts, 1-9 for custom prompts)
3. `prompt`: The prompt used in the submission.
4. `test_output`: The raw output from the system (including LLM output without processing).
5. `test_code`: Reformatted test code (restricted to 25,000 characters or less).

Submission Requirements:

- A fixed-prompt submission with `prompt_number` 0 is required.
- A custom prompt submission with `prompt_number` 1 is required.
- Additional custom-prompt submissions with `prompt_number` 2–9 are optional.

The `test_code` can be automatically, semi-automatically, or manually formatted from the `test_output` (see Section 6 for details).

To ensure `test_code` runs without errors, add `from genai_code_file import *\n` (or `from genai_code_file import primary_method_name>\n`) to the beginning of the `test_code` field.

An example submission is provided in Appendix (Section A.3).

4.2 System Descriptions

All participants are required to submit a system description. A system description should include, but is not limited to, the following information:

Section 1. Submission Identifier(s). List the submission IDs for each system output submitted.

Section 2. System Description. A brief technical description of the system and the system model used.

Section 3. System Hardware Description and Runtime Computation. Describe the computing hardware setup(s) and report the number of CPU and GPU cores. A hardware setup is the aggregate of all computational components used.

Section 4. Training Data and Knowledge Sources. List the resources used for system development and runtime knowledge sources, if any.

Section 5. Prompt Construction and Post-Processing. Describe how the custom prompts were constructed and what was done to post-process the LLM output into validly formatted test code.

Section 6. References. List pertinent references, if any.

5 Evaluation Metrics

For each test output, three metrics are measured or verified:

- **Correctness:** The test output executes without errors on the correct code (`code_correct`).
- **Error Detection:** The test output exposes errors in the incorrect code implementations (`code_incorrect_1` and `code_incorrect_t`).
- **Line Coverage:** The line coverage of the test output on the correct code (`code_correct`).

A test output is considered correct if it executes without errors on `code_correct`, or if it produces an error when testing the incorrect code. The following scores are calculated for each `(system, prompt_number)` pair:

1. Percentage of problems with correct tests.
2. Percentage of problems with correct tests that detect errors in `code_incorrect_1`.
3. Percentage of problems with correct tests that detect errors in both `code_incorrect_1` and `code_incorrect_t`.
4. Percentage of problems correct tests and detect errors in both `code_incorrect_1` and `code_incorrect_t` and achieve 100%-line coverage.
5. Mean line coverage percentage for correct tests.

Results will be reported separately for fixed prompt and custom prompt conditions. Additional metrics may be examined during post-evaluation analysis, potentially separating problems by category into “simple” and “extended.”

5.1 Software Used for Performance Metric Scoring

Programs are tested using the Python package `pytest`, and line coverage is generated using the Python package `coverage`. The testing environment will be Python 3.12 and will include all standard libraries (such as `os` and `typing`) and additional packages including `pytest`, `coverage`, `numpy`, `panadas` and all packages used in the provided input codes. Additional Python packages will be installed as needed on a case-by-case basis.

6 Protocol and Rules

Rules and Restrictions

- Participants are not allowed to use the test dataset for training, modeling, or tuning their algorithms.
- Participants may use publicly available data that complies with applicable laws and regulations to train their models.
- All machine learning or statistical analysis algorithms must complete training, model selection, and tuning prior to running their system on the GenAI test data.

Advertising and Endorsement

- Participants may not make advertising claims about their standing in the evaluation or claim NIST endorsement of their system(s).
- The following language from the U.S. Code of Federal Regulations (15 C.F.R. §200.113 (d)) must be respected: NIST does not approve, recommend, or endorse any proprietary product or proprietary material.

Reporting and Publication

- NIST may generate a report summarizing the system results with participant team names.
- NIST plans to anonymize the team names but reserves the right to publish results with the participating teams' names.
- Participants may publish or disseminate the charts unaltered and with proper reference to their source.

Custom Prompt Restrictions

- Participants may use any prompts they wish for the custom prompts but may not provide human-generated tests.
- Participants may implement the specification with code and provide their systems with their code to test.
- Participants will be asked to share how they constructed the custom prompt when providing system descriptions.

Post-Processing of LLM Output

- Participants are allowed to post-process their LLM output to conform to the required format of the `test_code` field.
- Participants may run a script to extract and reformat the tests or fix interpreter or compiler errors without changing the tests.
- The direct LLM output should be provided in the `test_output` field.
- An optional script, `extract_test_code_from_test_output.py`, is provided to automatically extract tests.

7 Agreement

All participants who wish to submit data generation outputs will be required to complete and sign an agreement included in a registration form before uploading their submissions. This registration form will be provided to participants during the registration process for the pilot.

8 Tentative Schedule

The schedule for the GenAI Code Pilot challenge is:

July 16, 2025	Evaluation Plan posted
July 23, 2025	Registration and Submissions open for Round-1
September 12, 2025	Registration and Submissions close for Round-1
September 26, 2025	Preliminary Results available for Round-1
December 5, 2025	Registration and Submissions open for Round-2
January 22, 2026	Registration and Submissions close for Round-2
January 29, 2026	Preliminary Results available for Round-2

References

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program Synthesis with Large Language Models, August 2021. URL http://arxiv.org/abs/2108.07732. arXiv:2108.07732 [cs].

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, July 2021. URL http://arxiv.org/abs/2107.03374. arXiv:2107.03374 [cs].

A Appendix: Summarizing the Process on a Single Code

We summarize the entire chain with segments of the json file for the dry-run problem "00001_add'.

First, we provide the relevant part of the json file for the input received for that problem. Second, we provide an example submission. Third, we provide the relevant components of the ground-truth key file that contains the held-back codes. Last, we show how this example submission is scored.

A.1 Human-Readable Summary of Example

A human-readable example input is below.

A.1.1 Specification:

```
def add(x: int, y: int) -> int:
    """
    Given two integers x, and y, return the sum of x and y. If either x
     ↪  or y is not
    an integer, raise a TypeError Exception.
    """
```

A.1.2 Fixed Prompt

```
We have python code that implements the following specification.

Specification:

def add(x: int, y: int) -> int:
    """
    Given two integers x, and y, return the sum of x and y. If either x
     ↪  or y is not
    an integer, raise a TypeError Exception.
    """


Please write python pytest test code that comprehensively tests the code
 ↪  for method add to determine if the code correctly meets the
 ↪  specification or not. When writing tests:
* write a comprehensive test suite,
* test edge cases,
* only generate tests that you are confident are correct, and
* include tests for TypeError cases.

Please write '###|=-=-=beginning of tests=-=-=|' before the tests. Write
 ↪  '###|=-=-=end of tests=-=-=|' immediately after the tests. Import any
 ↪  needed packages, including pytest. Additionally import the code being
 ↪  tested by adding the line `from genai_code_file import add` the line
 ↪  after '###|=-=-=beginning of tests=-=-=|'. Do not provide an
 ↪  implementation of the method add with the tests.
```

A.1.3 System Output

Human-readable example output for the example input is below.

```
from genai_code_file import add
import pytest

class TestCode(object):

    def test_add(self):
        assert add(2, 3) == 5
        with pytest.raises(TypeError):
            add("abc", 5)
```

A.2 Input

The relevant part of the input json provided is:


```
{
  "name": "Problem Inputs for Code Bank Pilot Dry-Run Problems",
  "version": "1.00",
  "Evaluation_Version": "Pilot1",
  "code_list": [
    {
      "trial_id": "00001_add",
      "testing_import_statement": "from genai_code_file import add",
      "primary_method_name": "add",
      "specification": "def add(x: int, y: int) -> int:\n    \"\"\"\n
      ↪  Given two integers x, and y, return the sum of x and y. If
      ↪  either x or y is not\n    an integer, raise a TypeError
      ↪  Exception.\n    \"\"\"\n",
      "prompt_fixed": "We have python code that implements the following
      ↪  specification.\n\nSpecification:\n\ndef add(x: int, y: int) ->
      ↪  int:\n    \"\"\"\n    Given two integers x, and y, return the
      ↪  sum of x and y. If either x or y is not\n    an integer, raise
      ↪  a TypeError Exception.\n    \"\"\"\n\n\nPlease write python
      ↪  pytest test code that comprehensively tests the code for method
      ↪  add to determine if the code satisfies the specification or
      ↪  not. When writing tests:\n* write a comprehensive test
      ↪  suite,\n* test edge cases,\n* only generate correct tests,
      ↪  and\n* include tests for TypeError cases.\n\nPlease write
      ↪  '###|=-=-=beginning of tests=-=-=|' before the tests. Write
      ↪  '###|=-=-=end of tests=-=-=|' immediately after the tests.
      ↪  Import any needed packages, including pytest. Import the code
      ↪  being tested by adding the line `from genai_code_file import
      ↪  add` the line after '###|=-=-=beginning of tests=-=-=|'. Do not
      ↪  provide an implementation of the method add with the tests."
    },
    ...
```

```
  ]
}
```

A.3 Example Submission

The segment of the submission json for this problem is:

```
{
  "name": "NIST Baseline Two Test Code Submission Pilot Dry-Run
   ↪  Problems",
  "version": "1.00",
  "system": "nist_baseline_two_test_code",
  "code_list": [
    {
      "trial_id": "00001_add",
      "prompt_number": "0",
      "prompt": "We have python code that implements the following
       ↪  specification.\n\nSpecification:\n\ndef add(x: int, y: int) ->
       ↪  int:\n    \"\"\"\n    Given two integers x, and y, return the
       ↪  sum of x and y. If either x or y is not\n    an integer, raise
       ↪  a TypeError Exception.\n    \"\"\"\n\n\nPlease write python
       ↪  pytest test code that comprehensively tests the code for method
       ↪  add to determine if the code satisfies the specification or
       ↪  not. When writing tests:\n* write a comprehensive test
       ↪  suite,\n* test edge cases,\n* only generate correct tests,
       ↪  and\n* include tests for TypeError cases.\n\nPlease write
       ↪  '###|=-=-=beginning of tests=-=-=|' before the tests. Write
       ↪  '###|=-=-=end of tests=-=-=|' immediately after the tests.
       ↪  Import any needed packages, including pytest. Import the code
       ↪  being tested by adding the line `from genai_code_file import
       ↪  add` the line after '###|=-=-=beginning of tests=-=-=|'. Do not
       ↪  provide an implementation of the method add with the tests.",
      "primary_method_name": "add",
      "test_output": "from genai_code_file import add\nimport
       ↪  pytest\n\nclass TestCode(object):\n    def test_add(self):\n
       ↪  assert add(2, 3) == 5\n        with pytest.raises(TypeError):\n
       ↪  add(\"abc\", 5)\n",
      "test_code": "from genai_code_file import add\nimport
       ↪  pytest\n\nclass TestCode(object):\n    def test_add(self):\n
       ↪  assert add(2, 3) == 5\n        with pytest.raises(TypeError):\n
       ↪  add(\"abc\", 5)\n"
    },
    {
      "trial_id": "00001_add",
      "prompt_number": "1",
```

```
      "prompt": "We have python code that implements the following
       ↪  specification.\n\nSpecification:\n\ndef add(x: int, y: int) ->
       ↪  int:\n    \"\"\"\n    Given two integers x, and y, return the
       ↪  sum of x and y. If either x or y is not\n    an integer, raise
       ↪  a TypeError Exception.\n    \"\"\"\n\n\nPlease write python
       ↪  pytest test code that comprehensively tests the code for method
       ↪  add to determine if the code satisfies the specification or
       ↪  not. When writing tests:\n* write a comprehensive test
       ↪  suite,\n* test edge cases,\n* only generate correct tests,
       ↪  and\n* include tests for TypeError cases.\n\nPlease write
       ↪  '###|=-=-=beginning of tests=-=-=|' before the tests. Write
       ↪  '###|=-=-=end of tests=-=-=|' immediately after the tests.
       ↪  Import any needed packages, including pytest. Import the code
       ↪  being tested by adding the line `from genai_code_file import
       ↪  add` the line after '###|=-=-=beginning of tests=-=-=|'. Do not
       ↪  provide an implementation of the method add with the tests.",
      "primary_method_name": "add",
      "test_output": "from genai_code_file import add\nimport
       ↪  pytest\n\nclass TestCode(object):\n    def test_add(self):\n
       ↪  assert add(2, 3) == 5\n        with pytest.raises(TypeError):\n
       ↪  add(\"abc\", 5)\n",
      "test_code": "from genai_code_file import add\nimport
       ↪  pytest\n\nclass TestCode(object):\n    def test_add(self):\n
       ↪  assert add(2, 3) == 5\n        with pytest.raises(TypeError):\n
       ↪  add(\"abc\", 5)\n"
    },
    ...
    }
  ]
}
```


A.4 Relevant Key Information

The relevant components of the key json file, which include the held-back codes, are:


```
{
  "name": "Key Inputs for Code Bank Pilot_Dry-Run Problems",
  "version": "1.00",
  "Evaluation_Version": "Pilot1",
  "code_list": [
    {
      "trial_id": "00001_add",
      "testing_import_statement": "from genai_code_file import add",
      "source": "HumanEval",
      "source_text": "HumanEval/53",
      "category": "simple",
      "primary_method_name": "add",
```

```
      "specification": "def add(x: int, y: int) -> int:\n    \"\"\"\n
       ↪  Given two integers x, and y, return the sum of x and y. If
       ↪  either x or y is not\n    an integer, raise a TypeError
       ↪  Exception.\n    \"\"\"\n",
      "code_correct": "def add(x: int, y: int) -> int:\n    if ((not
       ↪  isinstance(x, int)) or isinstance(x, bool)) or ((not
       ↪  isinstance(y, int)) or isinstance(y, bool)):\n        raise
       ↪  TypeError('Inputs x and y must both be integers.')\n    return
       ↪  x + y\n",
      "code_incorrect_1": "def add(x: int, y: int) -> int:\n    if ((not
       ↪  isinstance(x, int)) or isinstance(x, bool)) or ((not
       ↪  isinstance(y, int)) or isinstance(y, bool)):\n        raise
       ↪  TypeError('Inputs x and y must both be integers.')\n    return
       ↪  x + 2*y\n",
      "code_incorrect_t": "def add(x: int, y: int) -> int:\n    return x
       ↪  + y\n"
    },
    ...
  ]
}
```

A.5 Scoring the Submission

For ease of understanding, we use human-readable formatting of the above example here.

From the submission, we have one fixed prompt submission and a custom prompt submission.

For the fixed prompt, identified by `prompt_number` "0", we received the tests in the `test_code` field, which are:

```
from genai_code_file import add
import pytest

class TestCode(object):

    def test_add(self):
        assert add(2, 3) == 5
        with pytest.raises(TypeError):
            add("abc", 5)
```

For the custom prompt, identified by `prompt_number` "1" in this submission, we received the tests in the `test_code` field, which are:

```
from genai_code_file import add
import pytest

class TestCode(object):

    def test_add(self):
```

```
        assert add(2, 3) == 5
        with pytest.raises(TypeError):
            add("abc", 5)
```

In this case the tests are exactly the same. Hence, we will continue scoring this example using only one of the files.

First, we run the tests against the held-back correct code, which is:

```
def add(x: int, y: int) -> int:
    if ((not isinstance(x, int)) or isinstance(x, bool)) or
            ((not isinstance(y, int)) or isinstance(y, bool)):
        raise TypeError('Inputs x and y must both be integers.')
    return x + y
```

When running the code, we notice that `add(2,3)` is 5, `add("abc", 5)` raises a TypeError exception. Hence, all assert statements pass and pytest outputs that all tests are correct. Hence, in this example, the *tests are correct*. As the tests are correct, we run coverage to get the line code coverage and get a coverage of 33%.

Next, we run it against one of our held-back incorrect programs, which is `code_incorrect_1`:

```
def add(x: int, y: int) -> int:
    if ((not isinstance(x, int)) or isinstance(x, bool)) or
            ((not isinstance(y, int)) or isinstance(y, bool)):
        raise TypeError('Inputs x and y must both be integers.')
    return x + 2*y
```

In this case, pytests runs `add(2,3)` and gets the output of 8, which is not 5. Hence, the assertion fails and pytest fails on this incorrect code. Therefore, the tests *find this error*.

Last, we run it against our second held-back incorrect code, which is `code_incorrect_t`:

```
def add(x: int, y: int) -> int:
    return x + y
```

In this case, all the tests still pass in pytest, so these tests *miss this error*. (In this case, the "+" cannot add an integer and a string and thus launches a TypeError exception even without the initial TypeError checks. However, running `add("abc", "def")` would in this version concatenate the strings to "abcdef" when a TypeError should be raised.)

B Appendix: Reference Key Data Format

The ground-truth code bank, or key, is a json file, with a field `code_files` that is a list of trials, where each entry is a single trial containing a single code under test. All string fields will have quotes and newlines as escape characters and spaces as they appear. The fields for each trial are:

1. `trial_id`: the id of the trial. Example: `00001_add`
2. `testing_import_statement`: the import statement to include in the generated test code. Example: `from gen_ai_code_file import add`

3. `specification`: the text specification of the code under test. The specification may contain code, input and output examples, or both. Example:

   ```
   "def add(x: int, y: int) -> int:\n    \"\"\"\n    Given two integers
   ↪  x, and y, return the sum of x and y. If either x or y is not\n
   ↪  an integer, raise a TypeError Exception.\n    \"\"\"\n"
   ```

4. `primary_method_name`: the name of the primary method that is being tested Example: `"add"`.

5. `source`: If this code was taken from another source, the name of the source.

6. `source_text`: text containing metadata relevant to the source of the code, including the problem id of the source.

7. `lines_per_method`: The mean (average) number of lines of code in each method.

8. `num_methods`: The number of methods in the code under test.

9. `imports_used`: "yes" if this code under tests imports external libraries, "no" otherwise.

10. `category`: the code bank set that this code under test is in. This value is either "simple" or "extended" for this pilot.

11. `code_correct`: the version of the correct code.

12. `code_incorrect_1`: the version of the incorrect code 1. The code for code_incorrect_1 is the correct code with some error that happens for some validly-typed input.

13. `code_incorrect_t`: the version of the incorrect code t. The code for code_incorrect_t is always code_correct without many of the TypeError and ValueError exception checks.

The problem input file is formed from the key file by selecting the subset of fields "trial_id", "testing_import_statement", "primary_method_name", and "specification", and then using these fields to construct the "prompt_fixed" field in an automated way.

C Appendix: Fixed Prompt Construction

We generate the fixed prompt for the fixed prompt submissions as follows:

> We have python code that implements the following specification.
>
> Specification:
>
> `<specification>`
>
> Please write python pytest test code that comprehensively tests the code for method `<primary method name>` to determine if the code correctly meets the specification or not. When writing tests: * write a comprehensive test suite, * test edge cases, * only generate tests that you are confident are correct, and * include tests for TypeError cases.
>
> Please write '###|=-=-=beginning of tests=-=-=|' before the tests. Write '###|=-=-=end of tests=-=-=|' immediately after the tests. Import any needed packages, including

pytest. Additionally import the code being tested by adding the line 'from genai_code_file import <primary method name>' the line after '###|=-=-=beginning of tests=-=-=|'. Do not provide an implementation of the method <primary method name> with the tests.

where <primary method name> is the primary method being tested, <specification> is the specification of the code.

A complete prompt for the dry-run problem "00001_add" example is in Section C.1.

C.1 Example Fixed Prompts

We have provided example prompts for the dry-run test problem "00001_add'. The text has been lined wrapped.

The prompt, with the specification and the code for the input "00001_add", is:

```
We have python code that implements the following specification.

Specification:

def add(x: int, y: int) -> int:
    """
    Given two integers x, and y, return the sum of x and y. If either x
     ↪  or y is not
    an integer, raise a TypeError Exception.
    """


Write python pytest test code that comprehensively tests the code for
 ↪  method add to determine if the code correctly meets the specification
 ↪  or not. When writing tests:
* write a comprehensive test suite,
* test edge cases,
* only generate tests that you are confident are correct, and
* include tests for TypeError cases.

Write '###|=-=-=beginning of tests=-=-=|' before the tests. Write
 ↪  '###|=-=-=end of tests=-=-=|' immediately after the tests. Import any
 ↪  needed packages, including pytest. Additionally import the code being
 ↪  tested by adding the line `from genai_code_file import add` the line
 ↪  after '###|=-=-=beginning of tests=-=-=|'. Do not provide an
 ↪  implementation of the method add with the tests.
```

The number of lines provided is as is. Occasionally there is more whitespace than desired, but whitespace was added to be sure that different components were separated.