

COVERUP: Coverage-Guided LLM-Based Test Generation

Juan Altmayer Pizzorno
University of Massachusetts Amherst
Amherst, MA, USA
jpizzorno@cs.umass.edu

Emery D. Berger[†]
University of Massachusetts Amherst / Amazon
Amherst, MA, USA
emery@cs.umass.edu

Abstract—This paper presents COVERUP, a novel system that drives the generation of high-coverage Python regression tests via a combination of coverage analysis and large-language models (LLMs). COVERUP iteratively improves coverage, interleaving coverage analysis with dialogs with the LLM to focus its attention on as yet uncovered lines and branches. The resulting test suites significantly improve coverage over the current state of the art: compared to CODAMOSA, a hybrid LLM / search-based software testing system, COVERUP substantially improves coverage across the board. On a per-module basis, COVERUP achieves median line coverage of 81% (vs. 62%), branch coverage of 53% (vs. 35%) and line+branch coverage of 78% (vs. 55%). We show that COVERUP’s iterative, coverage-guided approach is crucial to its effectiveness, contributing to nearly half of its successes.

I. INTRODUCTION

Test generation tools aim to increase the coverage of the program under test. By generating tests that cover a wider range of scenarios and execution paths, they can help uncover bugs missed by manually written tests. Unfortunately, current approaches often fail to achieve substantial coverage.

Pynguin exemplifies the *search-based software testing* (SBST) approach [1]. Starting from randomly created test cases, SBST employs genetic algorithms to mutate the tests in an attempt to increase coverage. Unfortunately, its search process can get stuck. CODAMOSA addresses this problem by tracking Pynguin’s progress [2]; when it concludes that the search process has stalled, it prompts a large language model (LLM) for a test. It then uses that test to re-seed the SBST, allowing it to resume progress. While CODAMOSA greatly improves Pynguin’s performance and constitutes the previous state of the art, the resulting code coverage can still remain relatively low (e.g., 34% overall branch coverage; see Section IV for details).

This paper proposes COVERUP, a novel approach that drives LLM-based test generation with coverage analysis. COVERUP incorporates detailed coverage information into prompts customized to the current state of the test suite, focusing the LLM on code that lacks coverage. Once the LLM generates a set of tests, COVERUP executes them and measures the resulting coverage. If the LLM-generated tests fail to sufficiently improve coverage or fail to run altogether, COVERUP continues the dialogue with the LLM, requesting

changes to improve coverage or fix the error. In doing so, COVERUP in effect further refines and clarifies the prompt, resulting in tests that significantly improve coverage.

To our knowledge, COVERUP is the first test generation system to prompt an LLM for tests specifying the lines and branches lacking coverage and that iteratively refines that prompt using coverage analysis.

We evaluate COVERUP using CODAMOSA as a baseline, and on the same benchmark applications originally used to evaluate it. Using the state-of-the-art OpenAI’s GPT-4 Turbo LLM for both, we find that COVERUP yields higher coverage in 61% of the benchmarks, significantly increasing all coverage metrics, whether computed across all code or aggregated over the various benchmarks. COVERUP increases median line coverage from 62% to 81%, median branch coverage from 35% to 53% and median line+branch coverage from 55% to 78%. Across all code, COVERUP increases line coverage from 54% to 61%, branch coverage from 34% to 43%, and line+branch coverage from 49% to 57%. We show that COVERUP’s iterative, coverage-guided approach is key to its effectiveness, accounting for roughly half of its successes.

This paper makes the following contributions:

- It presents COVERUP, a novel system that drives the generation of high-coverage Python regression tests via a combination of coverage analysis and large-language models;
- It conducts an empirical analysis of COVERUP, showing that it significantly advances the state of the art.

II. RELATED WORK

Automated test generation is a well-established field of research. Among the various proposed methods are specification-based [3], random [4], [5], feedback-directed random [6], symbolic execution guided random (“concolic”) [7]–[9], search-based software testing (SBST) [1], [10]–[13] and transformer based [14] approaches.

The success of large language models on various tasks has motivated their application to software engineering; Wang et al. survey 102 recent papers using LLMs for software testing [15]. This section focuses on previous work most closely related to COVERUP.

TESTPILOT prompts an LLM to generate JavaScript unit tests based on the function under test’s implementation, its

[†]Work done at the University of Massachusetts Amherst.

documentation and usage snippets [16]. Like COVERUP, TESTPILOT checks the tests generated by the LLM and continues the chat to refine the prompt in case of errors, but it does not prompt based on coverage, nor does it refine the prompt in case the new tests do not improve coverage.

Bareiß et al. study the performance of the Codex LLM on Java test case generation, among other code generation tasks [17]. Its prompts contain the signature of the method under test, an example of test generation, and the body of the method; it discards any tests that do not compile. By contrast, COVERUP’s prompts are based on code segments lacking coverage and they explicitly request tests to improve it. COVERUP also continues the chat with the LLM in case of build failure, failing tests, or lack of coverage. As Section IV-D shows, this iterative dialogue is responsible for almost 50% of COVERUP’s successful test generation.

Vikram et al. discuss prompting LLMs based on API documentation to generate property-based tests for Python [18], [19]. COVERUP instead bases its prompting on the source code and coverage measurements and, while it accepts property-based tests if the LLM generates them, it does not explicitly request them.

TiCODER prompts LLMs to generate tests based on a natural language description of the intended functionality of code [20]. Rather than facilitate regression testing, however, TiCODER generates tests with the goal of clarifying and formalizing user intent.

FUZZ4ALL [21] uses two separate LLMs to *fuzz test* programs in various programming languages. The *distillation* LLM takes in arbitrary user input, such as documentation and code examples, and generates prompts for the *generation* LLM, which generates test inputs. After the initial user input distillation and prompt selection, FUZZ4ALL repeatedly employs the generation LLM, mutating its prompt to produce additional test inputs. While FUZZ4ALL and COVERUP both use LLMs to generate test cases, their approaches differ fundamentally: FUZZ4ALL generates test inputs based on documentation or examples, relying on an user-provided test oracle for testing; COVERUP instead works based on the source code and coverage measurements, looking to add tests that improve the test suite’s coverage.

CODAMOSA [2], mentioned in the introduction, prompts the Codex LLM for a test when its SBST algorithm gets stuck. Like COVERUP, CODAMOSA selects code to prompt for based on its (lack of) coverage, and the prompt contains source code. However, it does not use coverage information within the prompt itself, nor does it systematically prompt for tests for all code lacking coverage; it only prompts the LLM when necessary to help SBST make progress. CODAMOSA represents the previous state of the art in test case generation for Python, and Section IV uses it as the baseline for comparison to COVERUP.

III. TECHNIQUE

COVERUP first measures the code coverage of the existing test suite and splits up the source code into segments that

require more testing (Section III-A). It then prompts the LLM for tests for each of these segments, using a chat interface (Section III-B). As the LLM generates tests, COVERUP executes them, again measuring coverage. If the test does not compile, fails or lacks coverage, it continues the chat, pointing out any problems and requesting improvements (Section III-C). Finally, having processed all code segments, COVERUP checks the extended test suite, looking for any integration problems (Section III-D). Figure 1 provides a graphical overview of these steps.

A. Code Segmentation

COVERUP’s first step is to measure the code coverage of any pre-existing test suite; it uses SlipCover [22], a recently introduced coverage analyzer that has near-zero overhead, for that task. It then uses the AST of each source file missing coverage to identify code segments that can be used for prompting. The goal is to be able to provide to the LLM, with each prompt, a code excerpt that is intelligible, provides enough context, includes the lines or branches lacking coverage, and is as short as possible.

Keeping code segments short is important for several different reasons. Older OpenAI LLMs such as GPT-3.5 and GPT-4, as well as on-premises LLMs like Meta’s LLaMA have small input windows of up to a few thousand tokens, limiting the maximum size of the code segment. OpenAI’s most recent GPT-4 Turbo models allow for inputs of up to 128K tokens, but this limit applies not only to the initial prompt, but to the total number of tokens in the entire sequence of prompts and responses in the case of a continued chat. Finally, OpenAI charges at a per-token rate in both prompts and responses, so that it literally pays to be succinct.

Each code segment typically includes an entire function or method and all its contents. If a segment contains a method, COVERUP also includes a few lines of the original class definition to provide the LLM with context, but omits other methods and definitions; see Figure 2 for an example. In an effort to enhance code comprehension, COVERUP retains all comments in a segment, although removing them could make the segment more concise.

Concretely, to identify the code segments in a source file, COVERUP first computes a set of “interesting” lines: these are lines that either lack coverage, or that are the source or the destination in branches that lack coverage. For each of these lines, it then looks in the AST for a class, function or method object containing that line. If the object found is a class and it spans more lines than a configurable limit, COVERUP adds that class definition as segment context and recursively looks for another, smaller object containing the line. This limit is intended to guide COVERUP towards acceptable excerpt sizes, facilitating its use with language models with more limited input window sizes. Algorithm 1 describes this process in more detail.

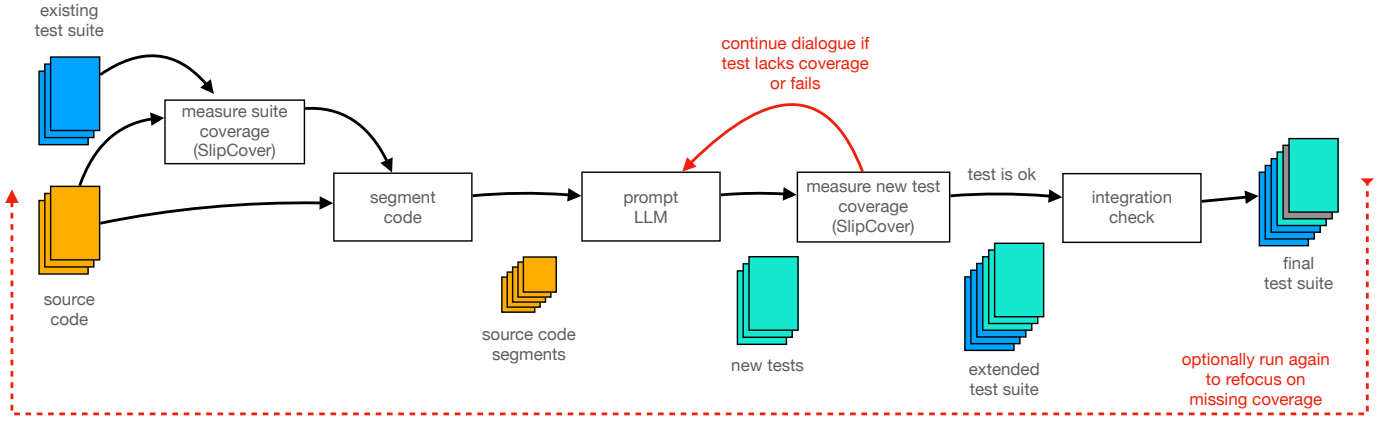


Fig. 1: **Overview: The execution steps of COVERUP.** After measuring the test suite’s coverage, COVERUP prompts the LLM so as to focus it on code segments that lack coverage. It checks each new test, either accepting it if it increases coverage or continuing the dialogue if it needs improvements. A final check helps ensure that the suite works as a whole. Given its incremental nature, once done COVERUP can be run again to refocus the LLM on any segments missed in previous passes.

Algorithm 1: Algorithm for identifying the code segments lacking coverage, based on a coverage measurement and a tentative maximum segment length.

```

Function IDENTIFY_SEGMENTS(coverage, max_len)
  code_segs ← ∅
  foreach file in coverage.files do
    interesting ←
      MISSING_LINES(coverage, file) ∪
      LINES_IN(MISSING_BRANCHES(coverage, file))

    ast ← PARSE_AST(file)
    foreach line in interesting do
      context ← empty list
      node ← FIND_LINE(ast, line)
      while (node is-a Class) and
        LENGTH(node) > max_len and
        (inner ← FIND_LINE(node, line)) do
        append node to context
        node ← inner
      code_segs ← code_segs ∪ {node, context}
  return code_segs

```

```

1 class AnsiTextWrapper(TextWrapper):
2   def _wrap_chunks(self, chunks: List[str]) -> List[str]:
3
4     lines = []
5     if self.width <= 0:
6       raise ValueError("invalid width %r (must be > 0)" \
7                        % self.width)
8     if self.max_lines is not None:
9       if self.max_lines > 1:
10        indent = self.subsequent_indent
11    ...

```

Fig. 2: **Example of a code segment with a class method:** COVERUP created a code segment for a class method, omitting over 200 lines of source code originally present between listing lines 1 and 2. The original code is from the `flutills` package.

B. Initial Prompting

COVERUP prompts the LLM for tests for every code segment it identifies; Figure 3 shows an example of an initial prompt. It has the following structure (circled numbers identify sections in the example):

- ① a statement assigning the LLM the *persona* [23] of an “expert Python test-driven developer”, intended to help guide it towards high quality tests;
- ② a sentence pointing out the code excerpt (segment),

identifying what file and module it comes from, and stating what lines or branches do not execute when tested. The portion specifying the lines and branches missing coverage is compressed using line ranges, simplifying the prompt and reducing token usage.

- ③ a request for a `pytest` test;
- ④ a series of other requests, such as “make sure that the new test is correct” and “include assertions” to steer the result towards usable tests;
- ⑤ a request that the response only include the new Python tests to facilitate extracting it from the response and to reduce token usage; and
- ⑥ the code segment with any relevant lines highlighted by prefixing them with their line numbers. Relevant lines are those that lack coverage or that are part of a branch that lacks coverage.

We found that tests generated by GPT-4 often included top-level code calling into `pytest.main` or into parts of the test itself. Figure 4 shows an example. While such top-level code

①

You are an expert Python test-driven developer.

②

The code below, extracted from `flutils/codecs/raw_utf8_escape.py`, module `flutils.codecs.raw_utf8_escape`, does not achieve full coverage: when tested, lines 19-20 do not execute.

③

Create a new pytest test function that executes these missing lines/branches, always making sure that the new test is correct and indeed improves coverage.

④

Always send entire Python test scripts when proposing a new test or correcting one you previously proposed. Be sure to include assertions in the test that verify any applicable postconditions. Please also make VERY SURE to clean up after the test, so as not to affect other tests; use 'pytest-mock' if appropriate. Write as little top-level code as possible, and in particular do not include any top-level code calling into `pytest.main` or the test itself.

⑤

Respond ONLY with the Python code enclosed in backticks, without any explanation.

⑥

```
```python
def _each_utf8_hex(text: _Str) -> \
 Generator[str, None, None]:
 for char in text:
 if ord(char) < 128 and char.isprintable():
 yield char
 else:
 continue
 utf8_bytes = char.encode('utf8')
 for utf8_byte in utf8_bytes:
 str_hex = '\\x%s' % hex(utf8_byte)[1:]
 yield str_hex
```
```

Fig. 3: Example of an initial prompt: The initial prompt selects a persona (1), identifies where the code comes from and states how it lacks coverage when tested (2), asks for tests (3-5), and shows the code segment (6). Any lines relevant to the request, i.e., used to identify missing coverage, are highlighted by prefixing them with their line numbers (as in “19:” in part (6)). The code shown is from the `flutils` package.

```
1 import pytest
2 from ansible.inventory.group import Group
3
4 def test_group_deserialize():
5     # Setup
6     group_data = {
7         ...
8     }
9     group = Group()
10
11     # Exercise
12     group.deserialize(group_data)
13
14     # Verify
15     assert ...
16     assert ...
17
18 # Register the test function for pytest
19 pytest.main(["-v", __file__])
```

Fig. 4: Excerpt of test calling to `pytest.main`: We found that certain tests generated by GPT-4 included top-level, unconditional calls into `pytest.main`. Such calls, executed when `pytest` loads the test file, caused it to restart the search for all tests in the suite, slowing it down immensely. Part (4) of the prompt now requests omitting such calls.

can make sense in a standalone test file, Python executes it as part of the loading process and doing so may greatly disrupt `pytest`’s operation. In fact, the calls in some of these early tests caused `pytest` to restart its test discovery, slowing it down to the point that it became unusable. For that reason, part ④ of the prompt includes instructions that the test not include such calls.

Prompt part ④ also instructs that the tests “clean up [...], so as not to affect other tests”. It also suggests the use of `pytest-mock`, a library useful for isolating the software under test from other parts of the code, facilitating test cleanup. Despite these instructions, we still observe test generations that include side effects; Section III-D describes how COVERUP detects and handles such tests.

As Figure 3 shows, in part ⑥ of the prompt, COVERUP prefixes certain lines with their numbers, such as in 20: before line 20. It adds such prefixes to all lines lacking coverage as well as to all lines that are part of a branch that lacks coverage. This improves the LLM’s understanding of the missing coverage: we found that prompts that only indicated the line number at the beginning of the excerpt led to more tests being generated that did not improve coverage.

If a code segment has no prior coverage, that is, every one of its lines lacks coverage, then COVERUP uses a slightly modified initial prompt where part ② simply states “the code does not execute”. Typically, the LLM will not generate enough tests to immediately yield high coverage in response to this simpler prompt, but this can be improved upon with a subsequent execution of COVERUP, where it refocuses the LLM on those lines missed in the first iteration.

C. Verification and Continued Chat

Once the LLM generates tests in response to the initial prompt, COVERUP executes them, again measuring coverage

①

This test still lacks coverage: line 615 and branches 603->exit, 610->608, 618->exit do not execute.

②

Modify it to correct that; respond only with the complete Python code in backticks.

Fig. 5: Example of a coverage follow-up prompt: COVERUP indicates to the LLM that a line and some branches weren't covered (1), asking that it correct the test (2).

①

Executing the test yields an error, shown below.

②

Modify the test to correct it; respond only with the complete Python code in backticks.

③

```
def test_AnsiTextWrapper_non_stripped_placeholder():
    wrapper = AnsiTextWrapper()
    wrapper.placeholder = " \x1b[31mplaceholder\x1b[0m
"
> assert wrapper.placeholder_len == len_without_ansi
(wrapper.placeholder.strip())
E      AssertionError: assert 13 == 11
E      + where 13 = <flutils.txtutils.AnsiTextWrapper
E      object at 0x7f992ca5e440>.placeholder_len
E      + and 11 = len_without_ansi(' \x1b[31
E      mplaceholder\x1b[0m')
E      + where ' \x1b[31mplaceholder\x1b[0m' = <built-
E      in method strip of str object at 0x7f992bc06c90>()
E      + where <built-in method strip of str object
E      at 0x7f992bc06c90> = ' \x1b[31mplaceholder\x1b[0m '.
E      strip
E      + where ' \x1b[31mplaceholder\x1b[0m ' = <
E      flutils.txtutils.AnsiTextWrapper object at 0
E      x7f992ca5e440>.placeholder

coverup-tests/tmp_test_s4dcdnes.py:7: AssertionError
```

Fig. 6: Example of an error follow-up prompt: COVERUP indicates to the LLM that an error occurred (1), requests a repair (2) and includes an excerpt of the error messages (3). In the original execution from which this example is taken, the LLM responds with an usable test (not shown).

using SlipCover. This step is made practical by SlipCover's near-zero overhead: using `coverage.py`, the only other alternative tool for Python, introduces a median of 180% overhead [22].

If the new tests pass and increase coverage, COVERUP saves them. But, if they do not increase coverage, or if they result in failures or errors, COVERUP continues the chat session, pointing out the problem(s) and requesting improvements. Figures 5 and 6 show examples of those prompts.

Prior to executing a new test, COVERUP looks for any Python modules used by the test that are absent from the system. These are typically test helper modules, such as the `pytest-ansible` plugin used to help test the `ansible` package, but could also be indicative of a module required by the program under test that has not yet been installed. COVERUP offers options to record the missing modules in a Python `requirements.txt` format to facilitate making them available for a subsequent run.

D. Integration Check

While COVERUP only saves tests that pass when ran by themselves, it is still possible that some of these do not clean up properly, or otherwise have side effects ("state pollution") that cause other tests to fail [24]. This issue is not unique to LLM-based test generation: Fraser et al. [25] report side effects on 50% of test classes generated by EVOSUITE [11], an SBST test generator. Figure 7 shows an example where a generated test not only modifies, but also, in a misguided attempt to clean up, deletes global symbols used by the code under test.

To attempt to detect and handle such tests, once COVERUP has processed all outstanding code segments, it executes the entire suite in a single session. If tests then fail, COVERUP attempts to identify the cause, disabling tests as necessary by renaming their files so that `pytest` ignores them. Such tests remain available for review and possible reactivation by the user, as they are known to contribute to coverage.

COVERUP supports two different methods for handling failing tests. It can simply disable all failing tests without attempting to ascertain the cause of the errors. This method is fast and allows the user to move on quickly to prompting for more tests, but may lead to a significant loss in coverage as various tests are disabled unnecessarily. Alternatively, COVERUP can look for the test or tests actually causing the failures by successively running subsets of the entire suite, an approach adapted from Delta Debugging [26]. This method requires additional test runs, and thus more time, but is likely to reduce the loss in coverage resulting from disabled tests.

E. Handling Flaky Tests

Flaky tests are those that can both pass or fail inconsistently, without any changes to the code under test [27], [28]. Their unreliable behavior has a variety of causes; in the context of LLM generated tests, they can also in part result from not providing the LLM sufficient information. Figure 8 shows a test for the `mimesis` data generator package. The code under test randomly assigns a blood type to its `Person` object; the LLM, not having been provided with the list of valid names, assumes that it includes 0- and 0+, when the actual code uses O- and O+ (using the letter O, not zero). Consequently, the test fails whenever that blood type is assigned.

While COVERUP does not attempt to detect flaky tests by default, it supports passing custom arguments to `pytest`. This approach lets the user specify that each test should be repeated a number of times using the `pytest-repeat` plugin. Flaky

```

1 import pytest
2 ...
3 from ansible import constants as C
4
5
6 @pytest.fixture
7 def setup_mocks(mock):
8     display_mock = mock.patch(
9         'ansible.utils.display.Display',
10         autospec=True)
11     ...
12
13
14 def test_process_include_results(setup_mocks):
15     ...
16
17     # Setting up the constants to be used in the test
18     C._ACTION_ALL_INCLUDES = ['include', 'include_tasks',
19                               'import_tasks',
20                               'import_playbook']
21     C._ACTION_INCLUDE = 'include'
22     C._ACTION_ALL_INCLUDE_TASKS = ['include_tasks',
23                                    'import_tasks']
24
25     ...
26
27     assert ...
28     assert ...
29
30     ...
31
32     # Clean up the constants
33     del C._ACTION_ALL_INCLUDES
34     del C._ACTION_INCLUDE
35     del C._ACTION_ALL_INCLUDE_TASKS

```

Fig. 7: Excerpt of a test with side effects: This LLM-generated test overwrites global constants and then, in a misguided attempt to clean up, deletes them. The test succeeds if executed by itself, but when executed along with other tests, the missing constants cause other tests to fail. COVERUP detects these failures during its integration check and optionally either disables all failing tests, or searches for the test causing the failures.

```

1 import pytest
2 from mimesis.providers.person import Person
3
4 @pytest.fixture
5 def person():
6     return Person()
7
8 def test_blood_type(person):
9     blood_type = person.blood_type()
10     # Assuming BLOOD_GROUPS is a constant defined in the
11     # mimesis.providers.person module
12     # If it's not accessible, we can define it here for
13     # the sake of the test
14     BLOOD_GROUPS = ['0-', '0+', 'A-', 'A+', 'B-', 'B+',
15                     'AB-', 'AB+']
16     assert blood_type in BLOOD_GROUPS

```

Fig. 8: Example of a flaky test: This test checks that the Person’s blood type, randomly assigned, is valid. It fails whenever the blood type is O+ or O−, as the LLM assumed these would be named using zero, rather than the letter O. By repeating the test a few times, COVERUP is likely to find the error and give the LLM a chance to correct it.

tests are then more likely to fail, leading COVERUP to give the LLM a chance to correct the problem.

F. Other Technical Challenges

Making COVERUP a practical tool poses several technical challenges.

One such challenge stems from the time spent in LLM inference and executing tests. Each individual prompt sent through OpenAI’s API typically requires several seconds to complete. Similarly, executing individual LLM-generated tests and measuring the new coverage achieved requires additional time. COVERUP repeats this process for each code segment as well as each time the dialogue is continued. Even though COVERUP limits the amount of time spent waiting on responses and test executions, if each code segment was processed serially, COVERUP would take an unacceptable amount of time creating tests for packages of nontrivial size. COVERUP instead prompts for tests and verifies them asynchronously, using the Python `asyncio` package. As a result, during COVERUP’s evaluation (Section IV) we observe a 500x speedup over serial execution.

Other practical challenges arise because OpenAI’s API is provided as a cloud service and subject to various limits. COVERUP handles a number of timeout and other error conditions automatically. To avoid exceeding the rate limits imposed by OpenAI, COVERUP spreads out its requests using a leaky bucket [29] scheme implemented by the Python module `aiolimiter`. Additionally, COVERUP can create and use checkpoint files, allowing the user to resume prompting for tests (and not lose any progress so far) after interrupting it or stopping due to unforeseen circumstances, such as the OpenAI account running out of funds.

IV. EVALUATION

Our evaluation investigates the following questions:

- RQ1:** How does COVERUP’s coverage compare to the previous state of the art? (Section IV-B)
- RQ2:** How does COVERUP’s coverage compare to CODAMOSA using a state-of-the-art LLM? (Section IV-C)
- RQ3:** How effective is COVERUP’s continued dialogue at increasing coverage? (Section IV-D)

A. Experimental Setup

Benchmarks: We utilize a benchmark suite collated by the authors of CODAMOSA [2], available at <https://github.com/microsoft/codamosa>; we refer to it as suite CM. It stems originally from 35 projects used in the evaluation of BugsInPy [30] and Pygoblin [31]. The suite is made up of a subset of the Python modules in these projects, selected to focus on cases where Pygoblin has difficulty achieving full coverage: after removing the modules that failed to produce results, removing the modules where Pygoblin achieved full coverage within one minute, and downsampling the number of modules with a common parent, the suite contains 486 benchmark modules from 27 projects.

Given suite CM’s focus, evaluating COVERUP only on it would leave open the question of how it performs on code where

```
--model_name gpt-4-1106-preview --temperature 0
--model-base-url https://api.openai.com
--model-relative-url /v1/chat/completions
--include-partially-parsable True
--allow-expandable-cluster True
--algorithm CODAMOSA
--uninterpreted_statements ONLY
```

Fig. 9: **Configuration for CODAMOSA (gpt4)**: This configuration selects the same LLM and temperature as COVERUP, but is otherwise based on CODAMOSA’s best performing configuration, `codamosa-0.8-uninterp`.

SBST, or more precisely, the original pre-CODAMOSA Pynguin already performed well. For that reason, to investigate RQ1 we create an additional suite from the same source, using only modules for which Pynguin achieved full coverage, naming it PY. Further, if a module is already in CM, indicating that Pynguin failed at least once to achieve full coverage, we exclude it from PY. By evaluating COVERUP on both suites, we obtain a fuller picture of COVERUP’s performance.

Baselines: We use two versions of CODAMOSA, the previous state of the art in automated case test generation, as baselines for evaluation on the CM suite: the original version that uses the Codex model (“CODAMOSA (codex)”) and our adapted version that uses the same model as COVERUP (“CODAMOSA (gpt4)”).

To investigate RQ1, we utilize the tests CODAMOSA (codex) originally generated using its best performing configuration (`codamosa-0.8-uninterp`), available from its “dataset” package at <https://github.com/microsoft/codamosa-dataset>. The dataset includes 16 runs of CODAMOSA for each of the benchmark modules. Rather than use Pynguin’s coverage measurements from when the tests were generated, which differs subtly from SlipCover in how it counts branch coverage, as we describe below, we re-execute its tests and measure the resulting coverage (Section IV-A).

For RQ2, we adapt CODAMOSA to use the same LLM as COVERUP. The `code-davinci-002` (“codex”) LLM used by the original CODAMOSA, which is no longer available, used *code completion* prompts, where the LLM, given a code fragment, responds with a suggestion on how to complete it. CODAMOSA (codex) sent that LLM portions of the code under test to provide it with context as well as the header of a unit test function for that code. We create CODAMOSA (gpt-4) by modifying it to use OpenAI’s chat API and to insert instructions requesting a code completion before CODAMOSA (codex)’s original prompt. Figure 10 shows an example of the resulting prompt.

We configure CODAMOSA (gpt4) to use OpenAI’s `gpt-4-1106-preview` LLM, setting a temperature of 0, the same employed by COVERUP in this evaluation, but otherwise leave the configuration identical to that of `codamosa-0.8-uninterp`. Figure 9 shows the details.

Metric: We utilize the line, branch and combined line and branch coverage as metrics, computing these over all benchmark modules. We also compute the combined line and

①

Complete the following unit test function. Only write the completion; do NOT rewrite the function. Do NOT include any comments or description.

②

```
from .primitive import Register

def mute(*objects: Register) -> None:
    """
    Use this function to mute multiple register-objects
    at once.

    :param objects: Pass multiple register-objects to the
    function.
    """
    err = ValueError(
        "The mute() method can only be used with objects "
        "that inherit from the 'Register class'."
    )
    for obj in objects:
        if not isinstance(obj, Register):
            raise err
        obj.mute()

[...]

# Unit test for function mute
def test_mute():
```

Fig. 10: **Example prompt after adapting CODAMOSA to a chat LLM**: The figure shows a CODAMOSA completion prompt, adapted for use with a chat LLM. The prompt prefixes instructions to complete the code (1) before including the original prompt (2). The code shown is from the `flutils` package.

branch coverage on a per-module basis. It is necessary to include both line and branch coverage as metrics because in Python, branch coverage does not subsume line coverage: various situations can lead the Python interpreter to throw exceptions, and when thrown, these exceptions are not recorded as branches in coverage information.

We measure all coverage using SlipCover [22], including those for CODAMOSA (codex) generated tests. We do not use Pynguin’s coverage measurements from when CODAMOSA (codex) generated them because Pynguin measures branch coverage differently from both SlipCover and Python’s de facto standard coverage tool, `coverage.py` [32]. Before executing a program, the Python interpreter compiles the source code to bytecode. Pynguin counts the jump instructions used to implement conditionals as branches and thus measures *clause* coverage [22], [33]. Figure 11 shows an example with a two-clause conditional and the resulting bytecode. Counting such jump instructions as branches increases the total number of possible “branches” and can thus lead to both inflated and understated coverage percentual numbers, depending on how many of those were followed during execution.

Execution Environment: We execute both the CODAMOSA tests and COVERUP using the `codamosa-docker` Docker image available at <https://github.com/microsoft/codamosa>, mod-

```

>>> dis.dis(compile("if A and B: ...", "", "exec"))
0          0 RESUME                               0

1          2 LOAD_NAME                             0 (A)
          4 POP_JUMP_FORWARD_IF_FALSE             4 (to 14)
          6 LOAD_NAME                             1 (B)
          8 POP_JUMP_FORWARD_IF_FALSE             4 (to 18)
         10 LOAD_CONST                             0 (None)
         12 RETURN_VALUE
>>         14 LOAD_CONST                             0 (None)
         16 RETURN_VALUE
>>         18 LOAD_CONST                             0 (None)
         20 RETURN_VALUE

```

Fig. 11: **Python bytecode for two-clause conditional:** Python compiles the two-clause conditional `if A and B` into bytecode that includes two jump instructions. If a coverage tool counts these as branches, it is really measuring clause coverage.

ified only to install SlipCover and to disable its default entryptpoint script, allowing easy execution of other scripts. The image is based on Debian 11 and includes Python 3.10.2, with which we run all benchmarks.

Prior to each measurement, our scripts install the benchmark-specific `package.txt` requirements file distributed with CODAMOSA using `pip`. Unfortunately, `pip` fails to install various of these requirements. Since these failures were originally ignored for CODAMOSA, we ignore them as well, in an effort to replicate the original conditions as closely as possible.

GPT-generated tests often assume that certain Python modules are available on the system. Before each COVERUP run, we install the modules `mock`, `pytest-datadir`, `pytest-mock`, `responses` and `hypothesis`. COVERUP supports automatically installing modules required by tests, but our evaluation does not use that functionality.

CoverUp options: We configure COVERUP to use OpenAI’s recent `gpt-4-1106-preview` LLM, setting its “temperature” to zero, and do not limit the number of output tokens. We leave COVERUP’s target code segment size (see Section III-A) at its default of 50 lines, and do not automatically repeat test executions to look for flaky tests (see Section III-E).

To place COVERUP on the same footing as CODAMOSA, we first run it without any pre-existing test suite. We then run it two more times, allowing it to build upon the test suite from the previous runs. We utilize COVERUP’s `disable failing test resolution` method that simply disables any tests that fail during integration check (see Section III-D), and do not pass any extra `pytest` flags (see Section III-E).

B. Comparison to the previous state of the art

To compare COVERUP to the previous state of the art, we first evaluate it on the CM suite, using CODAMOSA (codex) as the baseline. Figure 12 shows the line, branch and combined line and branch coverage obtained by COVERUP and the baseline over the entire benchmark suite; Figure 13 shows the median of those measurements on a module-by-module basis.

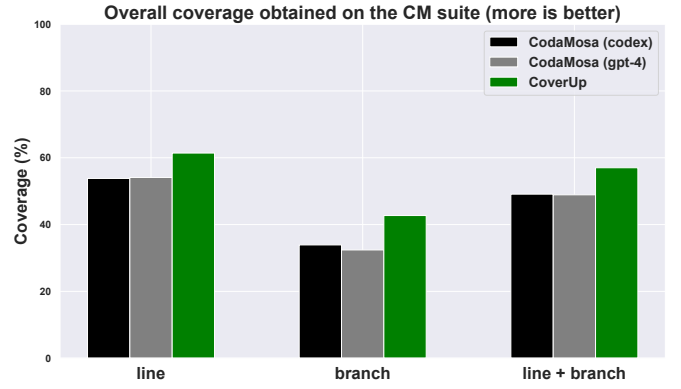


Fig. 12: **[RQ1, RQ2] COVERUP yields higher overall coverage:** COVERUP yields higher line, branch and line+branch coverage over the entire benchmark source than both CODAMOSA (codex), and CODAMOSA (gpt4).

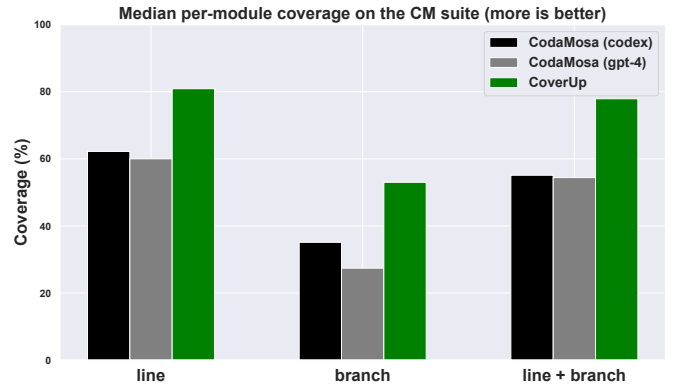


Fig. 13: **[RQ1, RQ2] COVERUP yields higher median per-module coverage:** COVERUP yields higher median line, branch and line+branch coverage on a module-by-module basis than than both CODAMOSA (codex) and CODAMOSA (gpt4).

Additionally, Figure 16 plots the difference between the combined line and branch coverage obtained by COVERUP and CODAMOSA on that suite, also on a module-by-module basis; green bars in the plot indicate modules where COVERUP achieved higher coverage.

As the figures show, COVERUP achieves higher line, branch and combined line and branch coverage than CODAMOSA (codex) both measuring over the entire benchmark code base and on a per-module basis.

We next evaluate the performance of COVERUP on code that the original Pyguitar already performed well. Figure 14 shows the results. Given COVERUP’s near 100% overall coverage and 100% median per-module coverage results, we conclude that COVERUP also performs well on such code.

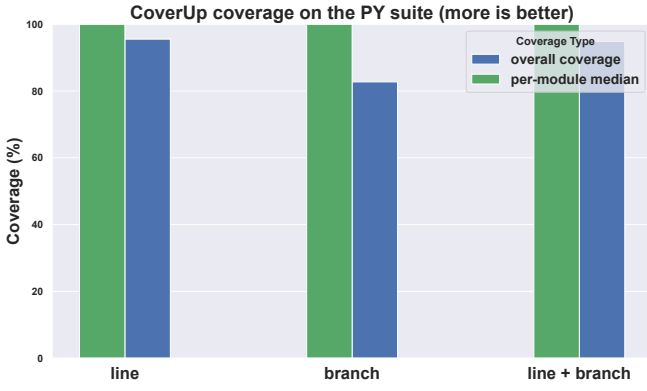


Fig. 14: **[RQ1] COVERUP coverage on the PY suite:** COVERUP yields near 100% overall and 100% median coverage on all metrics, showing that its effectiveness is not limited to the modules selected for the CM suite.

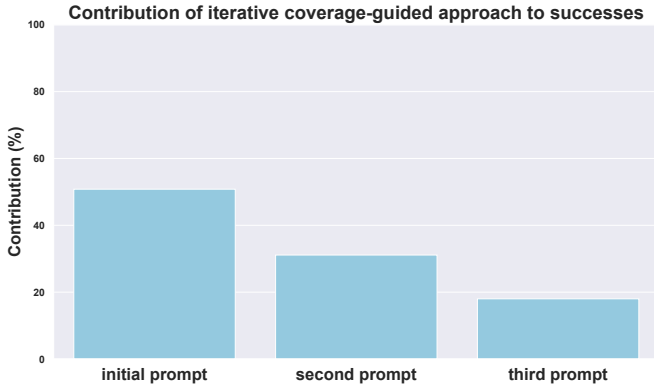


Fig. 15: **[RQ3] Contribution of iterative coverage-guided approach:** almost half (42.9%) of the successes were obtained by iterating and providing updated coverage information, demonstrating its importance.

[RQ1] Summary: COVERUP achieves significantly higher coverage than CODAMOSA (codex), the previous state of the art.

C. How does COVERUP’s coverage compare to CODAMOSA using a state-of-the-art LLM?

To investigate RQ2, we run CODAMOSA (gpt4) on the CM suite and measure the coverage it obtains. Figures 12 and 13 show the results, measuring coverage over the entire benchmark suite and on a per-module basis.

As the figures show, COVERUP also outperforms CODAMOSA (gpt4), achieving higher coverage both across the entire suite and a per-module basis. We also observe that CODAMOSA (gpt4)’s performance differs little from that of CODAMOSA (codex), with overall coverage and per-module line and combined line and branch measurements within 2% of each other.

[RQ2] Summary: COVERUP still outperforms CODAMOSA using a state-of-the-art LLM.

D. How effective are COVERUP’s continued dialogue?

To investigate RQ3, we process COVERUP’s logs generated while evaluated on the CM suite, identifying each successful test (i.e., which passes and improves coverage) that was generated immediately upon the first prompt, after continuing the chat with a second prompt, or after a third prompt. By default, and also in our evaluation, COVERUP continues the conversation for at most two additional prompts. Figure 15 shows the percentage of successes after the first, second and third prompts. As the figure shows, while the success rate decreases with each iteration, almost half of the successes (49.2%) were obtained through iterative refinement of the prompt, demonstrating its importance.

[RQ3] Summary: continuing the chat contributes to nearly half of successes, demonstrating its effectiveness.

V. THREATS TO VALIDITY

Benchmark selection: We utilize the same benchmark suite as CODAMOSA to facilitate a direct comparison to the state of the art. While so far our experience executing COVERUP to generate tests for other software has yielded similarly high coverage, selecting a different set of benchmarks could yield different results.

Execution environment: CODAMOSA’s original evaluation environment failed to install various Python modules that are prerequisites to the applications used as benchmarks. In an effort to replicate the original conditions as closely as possible, we ignored these failures as well. It seems likely that both CODAMOSA and COVERUP would be better able to generate tests if these modules were not missing. For COVERUP, the modules contributed by the `ansible` package are the most affected, with 326 test generation attempts being aborted due to missing modules.

LLM model dependency: All of COVERUP’s development and evaluation was performed using OpenAI GPT-4 and GPT-4 Turbo models. While COVERUP’s test execution and coverage measurement portions are independent of the LLM, its ultimate performance naturally depends heavily on the model’s ability to interpret its prompts and generate tests as requested.

VI. DISCUSSION AND FUTURE WORK

In an attempt to guide the LLM towards useful tests, COVERUP requests that tests contain assertions in its initial prompt (Section III-B) and also treats as an error any tests that fail (Section III-C). It is still possible, however, that the assertions generated by the LLM are not sufficient to verify the test postconditions. Even though this paper focuses on case generation and coverage, rather than the test oracle, we examined a sample of the assertions in the generated tests. Many of the assertions seemed appropriate, verifying that the code under test had the intended effect, but others, for

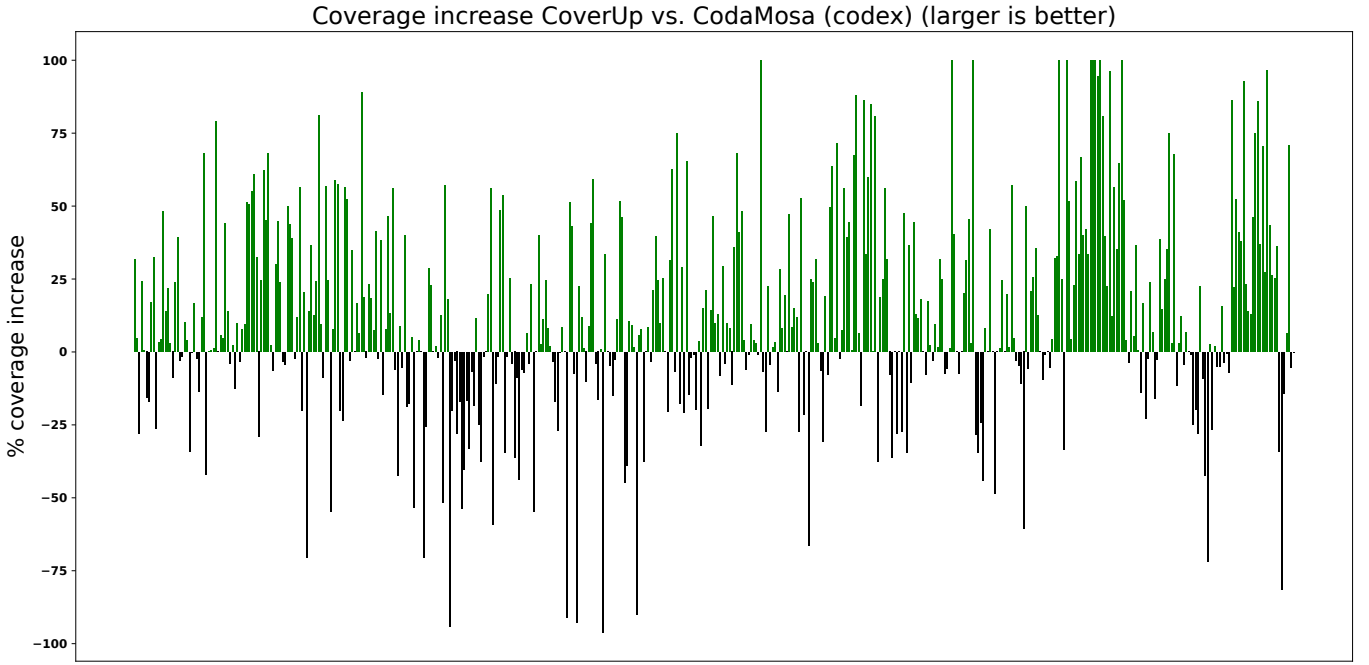


Fig. 16: [RQ1] **COVERUP yields higher coverage than the state of the art:** The graph shows the difference between the lines + branches coverage achieved by COVERUP and CODAMOSA (codex) for the various benchmark modules in the CM benchmark suite. Green bars indicate modules where COVERUP achieved a higher coverage.

example, merely checked that the state had changed. In future work, we intend to extend COVERUP to evaluate the assertions, possibly employing mutation testing [10], [34], and continuing the dialogue (Section III-C) to request improvements.

Another case that is likely to be a good candidate for continued prompting is when the LLM generates a test requiring Python modules that are not present on the system. As we describe in Section III-C, COVERUP detects this situation and optionally attempts to install or records a list of missing modules for later installation; it could, however, instead prompt the LLM to adjust or re-formulate the test so that the module is not necessary. Initial experiments suggest that this may be a viable way to automatically resolve such cases.

A number of initial prompt test generation failures could be attributed to COVERUP not providing enough context in its prompts, such as related definitions from other source files. In future work, we intend to experiment with ways to provide additional context. In particular, OpenAI’s *function calling* feature, which allows a model to request additional information in response to a prompt, may work well for this purpose, reducing the need for continued prompting.

Finally, we developed and evaluated COVERUP exclusively using OpenAI models (GPT-4, GPT-4 Turbo); we intend to adapt COVERUP to work with other models as well. We believe that adapting it for use with on-premises LLMs might be of special interest to industry practitioners, as it would help protect intellectual property.

| test case generator | line | branch | line + branch |
|---------------------|---------------|---------------|---------------|
| COVERUP | 61.4 % | 42.7 % | 57.0 % |
| CODAMOSA (codex) | 53.8 % | 33.9 % | 49.1 % |
| CODAMOSA (gpt4) | 54.1 % | 32.4 % | 48.9 % |

TABLE I: [RQ1, RQ2] Overall coverage obtained on the CM suite.

| test case generator | line | branch | line + branch |
|---------------------|---------------|---------------|---------------|
| COVERUP | 80.9 % | 53.0 % | 77.9 % |
| CODAMOSA (codex) | 62.2 % | 35.1 % | 55.1 % |
| CODAMOSA (gpt4) | 60.0 % | 27.4 % | 54.4 % |

TABLE II: [RQ1, RQ2] Median per-module coverage obtained on the CM suite.

| metric | line | branch | line + branch |
|-------------------|---------|---------|---------------|
| overall coverage | 95.5 % | 82.7 % | 94.8 % |
| per-module median | 100.0 % | 100.0 % | 100.0 % |

TABLE III: [RQ1] COVERUP coverage on the PY suite.

VII. CONCLUSION

This paper presents COVERUP, a novel approach for guiding LLM-based test generation with coverage analysis. We show that coupling coverage information with prompting, and iteratively refining prompts with updated coverage information to focus the LLM’s attention on improving uncovered code, is effective at generating test suites with higher coverage than past approaches.

| prompt | contribution |
|---------|--------------|
| initial | 50.8 % |
| second | 31.1 % |
| third | 18.0 % |

TABLE IV: [RQ3] Contribution of iterative coverage-guided approach to successes.

DATA AVAILABILITY STATEMENT

COVERUP is available as source code at <https://github.com/plasma-umass/coverup>.

REFERENCES

- [1] S. Lukaszczuk and G. Fraser, "Pynguin: automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '22. ACM, May 2022. [Online]. Available: <http://dx.doi.org/10.1145/3510454.3516829>
- [2] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 919–931.
- [3] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on java predicates," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 123–133. [Online]. Available: <https://doi.org/10.1145/566172.566191>
- [4] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, p. 32–44, dec 1990. [Online]. Available: <https://doi.org/10.1145/96267.96279>
- [5] A. Fioraldi, A. Mantovani, D. Maier, and D. Balzarotti, "Dissecting American Fuzzy Lop: A fuzzbench evaluation," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, mar 2023. [Online]. Available: <https://doi.org/10.1145/3580596>
- [6] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 2007, pp. 75–84.
- [7] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 213–223. [Online]. Available: <https://doi.org/10.1145/1065010.1065036>
- [8] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: Association for Computing Machinery, 2005, p. 263–272. [Online]. Available: <https://doi.org/10.1145/1081706.1081750>
- [9] N. Tillmann and P. de Halleux, "Pex - white box test generation for .NET," in *Proc. of Tests and Proofs (TAP'08)*, ser. LNCS, vol. 4966. Springer Verlag, April 2008, pp. 134–153. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/pex-white-box-test-generation-for-net/>
- [10] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2012.
- [11] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 416–419. [Online]. Available: <https://doi.org/10.1145/2025113.2025179>
- [12] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.
- [13] —, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2018.
- [14] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," 2021.
- [15] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," 2024.
- [16] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2024.
- [17] P. Bareiß, B. Souza, M. d'Amorim, and M. Pradel, "Code generation tools (almost) for free? A study of few-shot, pre-trained language models on code," 2022.
- [18] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 268–279. [Online]. Available: <https://doi.org/10.1145/351240.351266>
- [19] V. Vikram, C. Lemieux, and R. Padhye, "Can large language models write good property-based tests?" 2023.
- [20] S. K. Lahiri, S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, M. Musuvathi, P. Choudhury, C. von Veh, J. P. Inala, C. Wang, and J. Gao, "Interactive code generation via test-driven user-intent formalization," 2023.
- [21] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," *Proc. IEEE/ACM ICSE*, 2024.
- [22] J. A. Pizzorno and E. D. Berger, "Slipcover: Near Zero-Overhead Code Coverage for Python," in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*, 2023.
- [23] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with ChatGPT," 2023.
- [24] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: detecting state-polluting tests to prevent test dependency," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 223–233. [Online]. Available: <https://doi.org/10.1145/2771783.2771793>
- [25] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using EvoSuite," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, dec 2014. [Online]. Available: <https://doi.org/10.1145/2685612>
- [26] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, p. 253–267, oct 1999. [Online]. Available: <https://doi.org/10.1145/318774.318946>
- [27] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, oct 2021. [Online]. Available: <https://doi.org/10.1145/3476105>
- [28] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: <https://doi.org/10.1145/3428270>
- [29] J. Turner, "New directions in communications (or which way to the information age?)," *Comm. Mag.*, vol. 24, no. 10, p. 8–15, oct 1986. [Online]. Available: <https://doi.org/10.1109/MCOM.1986.1092946>
- [30] R. Widayarsi, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh, "BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '20. ACM, Nov. 2020. [Online]. Available: <http://dx.doi.org/10.1145/3368089.3417943>
- [31] S. Lukaszczuk, F. Kroiß, and G. Fraser, "An empirical study of automated unit test generation for python," *CoRR*, vol. abs/2111.05003, 2021. [Online]. Available: <https://arxiv.org/abs/2111.05003>
- [32] N. Batchelder, "nedbat/coveragepy: The code coverage tool for Python," <https://github.com/nedbat/coveragepy>, 2024.
- [33] N. Li, X. Meng, J. Offutt, and L. Deng, "Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report)," in *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*. IEEE Computer Society, 2013, pp. 380–389. [Online]. Available: <https://doi.org/10.1109/ISSRE.2013.6698891>
- [34] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.