TestForge: Feedback-Driven, Agentic Test Suite Generation

Kush Jain
Carnegie Mellon University
United States

Claire Le Goues
Carnegie Mellon University
United States

Abstract

Automated test generation holds great promise for alleviating the burdens of manual test creation. However, existing search-based techniques compromise on test readability, while LLM-based approaches are prohibitively expensive in practice. We present TestForge, an agentic unit testing framework designed to cost-effectively generate high-quality test suites for real-world code. Our key insight is to reframe LLM-based test generation as an iterative process. TestForge thus begins with tests generated via zero-shot prompting, and then continuously refines those tests based on feedback from test executions and coverage reports. We evaluate TestForge on TestGenEval, a real world unit test generation benchmark sourced from 11 large scale open source repositories; we show that TestForge achieves a pass@1 rate of 84.3%, 44.4% line coverage and 33.8% mutation score on average, outperforming prior classical approaches and a one-iteration LLM-based baseline. TestForge produces more natural and understandable tests compared to state-of-the-art search-based techniques, and offers substantial cost savings over LLM-based techniques (at $0.63 per file). Finally, we release a version of TestGenEval integrated with the OpenHands platform, a popular open-source framework featuring a diverse set of software engineering agents and agentic benchmarks, for future extension and development.

1 Introduction

Software testing plays a crucial role in software development; robust code bases typically include extensive unit and integration tests. However, creating effective tests is costly and time consuming [5, 6], causing developers to frequently ignore and neglect testing. As a result, there has been research into developing techniques for automated test generation [3, 8, 16, 18, 50, 54].

Classical test generation approaches like EvoSuite [18] and Pynguin [30] focus primarily on improving test coverage using search-based techniques (e.g., genetic programming, random search). These methods are capable of producing relatively high-coverage test suites. However, they also often generate suites that are difficult to understand and maintain [40]. The resulting tests require substantial developer effort for debugging and comprehension. This, in turn, hinders overall adoption [8].

Meanwhile, recent advances in Large Language Models (LLMs) have demonstrated remarkable capabilities to generate human-readable code [4, 19, 29, 34]. Tools like Copilot or Cursor have proven particularly effective for code generation, enhancing developer productivity [1, 2]. That said, off-the-shelf LLMs perform relatively much less well on test generation tasks as compared to code generation for specific functionality; one reason may be the specific context required to write effective tests [22, 38, 43]. Generating tests for large scale projects requires reasoning about the code under test, along with project and library dependencies used in the code under test. This limits prior approaches that only take in limited context such as the method or file under test [32, 43, 46]. Furthermore, LLMs struggle with hallucination [22, 26, 28]; hallucinated test setup and outputs often cause generated tests to fail, hindering adoption. That said, LLMs have demonstrated potential to improve classical techniques [25] by boosting coverage in scenarios where genetic programming plateaus.

A particularly promising recent direction for overcoming some of these limitations of LLM-based test generation takes an *agentic* approach. Agentic AI systems are characterized by the use of independent LLM-querying "agents" that have some degree of autonomy in choosing how to tackle a given task, and can interact with an environment providing access to additional tools like code search, static/dynamic analysis, or test execution. Such a system can autonomously explore code, edit tests, and self-reflect on its outputs. Recent techniques like CoverUp [41] and HITS [53] have demonstrated the value of incorporating dynamic information (like coverage analysis) or more complex slicing techniques into an LLM-mediated test generation loop. These approaches are better at generating both passing and high coverage test suites compared to both classical and one-iteration LLM techniques, as they can iterate based on rich execution feedback. However, while previous agentic methods have demonstrated the benefits of iterative feedback, their applicability has been limited by high costs—over two dollars per large file—when applied to complex, real-world code bases [22].

In this paper, we present TestForge, an agentic approach for test generation that can cost-effectively generate unit tests for large files in complex code bases. TestForge is predicated on three key insights. First, we frame test generation as an *iterative* process. TestForge begins by zero-shot prompting an underlying LLM model with the code under test. It then progressively refines those tests over multiple iterations to target undercovered lines, or regions exhibiting low mutation scores. Second, TestForge leverages detailed execution feedback—including compilation errors, runtime failures, and uncovered code segments—as an integral part of its agentic loop. We provide the full set of lines missing coverage as input to agent rather than selecting a subset as in prior approaches. This enables the agent to plan out multiple test cases in one iteration, improving efficiency. This feedback-driven process both improves test quality, and contributes to a high empirical pass@1 rate in our evaluation. Finally, TestForge crucially operates at the file level, rather than the individual method level considered in previous LLM- or agentic-based approaches. This dramatically improves cost-efficiency—the average file in our benchmark includes 58 methods—without in practice compromising effectiveness.

We evaluate TestForge's full test-suite-generation ability on the recently-released TestGenEval [22], a benchmark for evaluating test generation over complex real-world systems. For search-based techniques, we compare against Pynguin [30] (pure genetic programming) and CodaMosa [25] (genetic programming augmented

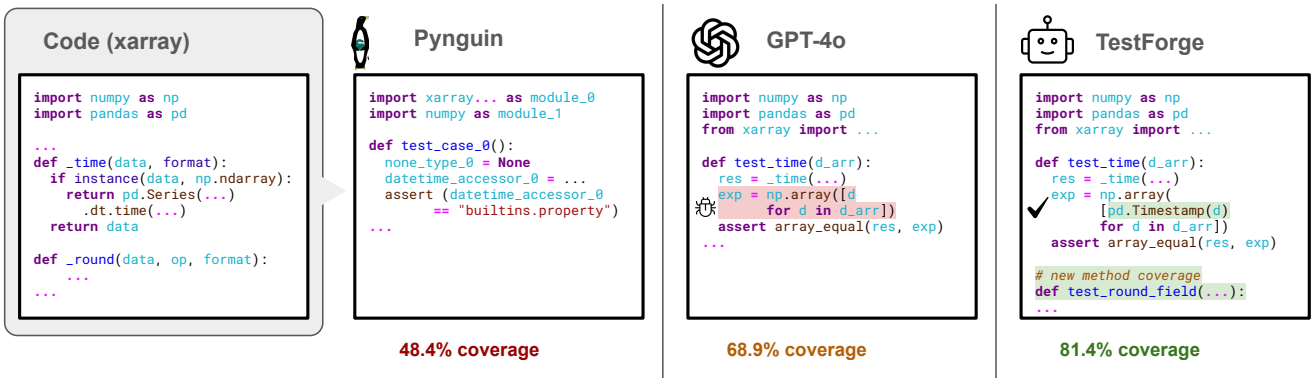


Figure 1: Example of tests generated by different approaches for timing and rounding functionality in pydata/xarray. GPT-4o generates a buggy test, which TestForge fixes, while also adding additional coverage improving tests.

with LLMs). We also compare against non-agentic LLM-based baselines CAT-LM [43], and GPT-4o [36]. Our experiments on the TestGenEval benchmark show that TestForge achieves a record pass@1 rate of 84.3%, a line coverage of 44.4%, and a mutation score of 33.8% outperforming both LLM baselines and existing genetic programming approaches on the programs where the techniques apply. The differences in mutation score over baselines are even more pronounced than coverage differences; TestForge achieves a mutation score improvement of 15.4% over our one-iteration baseline, improving the ability of generated test suites to catch synthetic bugs even when coverage is high. Because TestForge relies fundamentally on an LLM for code generation, the resulting tests are much more natural than those produced by classical search-based techniques. TestForge only costs $0.63 to generate tests for a file in our dataset.

Additionally, first, we integrate TestForge into the OpenHands[1] open source platform for developing and evaluating autonomous agents; it empowers researchers to build modular, agentic while facilitating reproducible research in the field of AI-powered software testing. Second, as part of our evaluation, we integrate the TestGenEval benchmark into OpenHands as well, to better support reproducibility and future research.

In summary, our contributions are as follows:

- TestForge, a state-of-the-art and cost-effective test generation agentic system. TestForge uses dynamic feedback and an interative approach to generate high-coverage and effective test suites for real-world code.
- Empirical results demonstrating TestForge's effectiveness compared to both classical and LLM-based baselines, in terms of pass@1 rate, coverage and mutation scores.
- An ablation study of our design decision to start from the zero-shot generated test suite and an experiment measuring line coverage compared to the number of iterations of TestForge.
- An integration of both TestForge and the TestGenEval benchmark with OpenHands, supporting reproducibility and extension, and evaluation of any new test generation system built with the OpenHands framework on the real-world benchmark.[2]

The rest of this paper is organized as follows: Section 2 provides an example showcasing key insights behind TestForge, Section 1 outlines the full design of TestForge, while Section 4 and Section 5 outline our evaluation setup and results respectively.

2 Illustrative Example

We begin by motivating and illustrating our approach with an example. The left-hand-side of Figure 1 shows a code snippet from `pydata/xarray`,[3] a widely used Python package built on top of numpy that enables labeling and aggregation over multi-dimensional tensors. TestForge generates tests at the file-level; that is, it aims to unit test all 24 methods in a file. The code under test in this file includes a number of utility accessor methods, such as rounding an array of objects, or converting a date object to a string time.

The second column of Figure 1 shows a test for this file produced by Pynguin [30], a search-based approach that uses a genetic programming heuristic to generate high-coverage unit tests for Python. Pynguin achieves 48.4% line coverage and 0.9% mutation score with a 10 minute compute budget on this file using four generated tests; the Figure shows the first such generated test. In addition to low coverage and near zero mutation score for the generated suite, note, crucially, that this test is difficult to understand. Methods and variables receive generic names, and the assertion is, plainly, difficult to follow and unrelated to the code under test (purpose is to assert the type of the instantiated variable). We observe this pattern with other tests generated in our evaluation. Additionally, for this file, of four tests Pynguin generates, three fail due to issues with test inputs, such as trying to round a null variable or run greatest common denominator on a string. Pynguin cannot fix such tests, and instead simply (though correctly) marks them as expected to fail.

Simply and directly zero-shot prompting GPT-4o [36] with the code under test, and requesting unit tests for the file, results in 18 tests. These tests achieve a coverage of 68.9% and mutation score of 47.4% on the file, outperforming Pynguin; the names and structure

[1] https://github.com/All-Hands-AI/OpenHands; Note that we omit a link to the pull request for anonymous review.

[2] Anonymized replication: https://anonymous.4open.science/ r/OpenHands-7E28/; we will link github repositories and pull requests in a final version of this paper.

[3] https://github.com/pydata/xarray

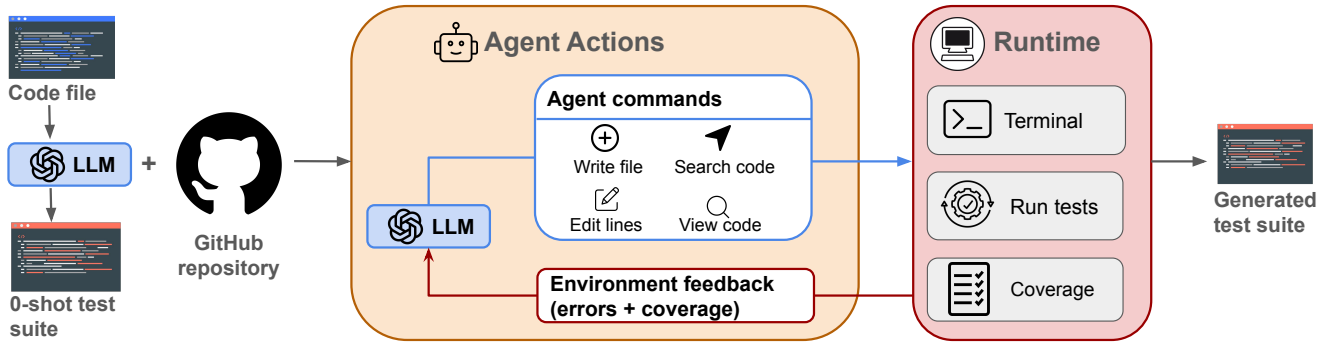


Figure 2: Overview of TestForge. We start by generating a zero-shot test suite and allowing our agent to interact with the repository with the generated test suite. We include the ability to search code, view code, write and edit files along with environment capabilities to run commands and tests. The output is a full test suite for the code file under test.

of this test better match our expectations for human-readable code. Despite improved coverage and mutation score, however, there are still methods in `pydata/xarray` that this one-iteration suite does not execute, like the `_round` method. Additionally, hallucination and incorrectness are well-known problems with LLM-generated code, and GPT-4o generated tests frequently contain subtle bugs. Here, for example, the output `exp` checked in the assertion should be converted to a timestamp before the comparison. Furthermore, the issue is clear, with Python raising an `AttributeError` when running the test and even displaying the expected output with the time series data type as part of the message. The output from executing the test can therefore be easily leveraged to improve the overall test suite (one of the core insights behind TestForge).

Other LLM agent approaches such as HITS [53] and CoverUp [41] use dependency analysis and program slicing as part of their agentic loop. For each method, these approaches generate tests and try to refine them individually to maximize coverage over all methods. While this works well for cases where source files are reasonably short and self-contained, it unfortunately does not scale well to projects with long contexts and complex dependencies. The full set of dependencies for a project such as `pydata/xarray` exceeds the 128k context window of most LLMs. This makes generating tests with both approaches very costly (over 3X the cost of TestForge), and thus not practical for these long context files; it is important to refine multiple tests at once rather than each test individually in order to save cost.

These observation motivate our approach in TestForge. TestForge starts with the zero-shot solution as a template for iterative test refinement. These generated tests are generally both readable, and achieve moderate coverage on the code under test. This allows us to frame test improvement as an iterative process, with TestForge slowly improving the coverage and correctness of the generated tests. Second, TestForge leverages execution feedback such as the missing lines in the coverage report and the test execution output. This allows the model to see and iteratively fix bugs in the initial generated tests. In this example, for example, TestForge can use the runtime error from the zero-shot test to fix the subtle bug and the missing lines in the coverage report for the `_round` method to add a new test that covers this method. In addition, the coverage report information provides targeted lines for the model to target, resulting in higher coverage of the refined test suites. The fourth column of Figure 1 shows the first test in the suite produced by TestForge for this example; it both fixes the bug in the GPT-4o tests, and the overall test suite covers previously untested methods, resulting in both a high coverage of 81.4% and high mutation score of 54.4% over the tested file. TestForge adds an additional 16 tests to the 18 GPT-4o-generated tests, with a total of 34 tests in the final test suite.

3 TestForge

In this Section, we provide an overview of TestForge, including the technical design details and the general agentic workflow. Section 3.1 shows the design of TestForge and an illustrative example of the agentic workflow. Section 3.2 provides more technical details behind each tool call our agent can make, Section 3.3 provides an overview of all execution feedback we can get from the environment, and Section 3.4 provides details about our implementation of TestForge and integration with OpenHands.

3.1 Overview

Figure 2 shows a full overview of TestForge. The input of TestForge is a code file to be tested; the output is a unit test suite for that file. Prior to the agentic loop, we prompt the LLM with the code under test, and ask it to generate an initial test suite, which we use as the starting test file (we show this is better than starting from scratch in our ablations (Section 5.3). Following this, TestForge enters the agentic loop. TestForge can either perform an *agent command* (actions listed in the orange box, labeled "Agent Actions") or *interact with the environment* (actions listed in the red box, labeled "Runtime"). When TestForge is finished (signified by executing the bash command `exit` or 25 iterations have elapsed, we save the generated test suite.

We provide a list of available agent actions and their corresponding input and output formats as tools when calling OpenAI's API. Our actions are consistent with CodeAct [52], as we build off of the CodeAct agent. If an action is malformed, we provide the agent with feedback corresponding to the command output (if the command fails altogether, we provide the tool parsing error). We follow

a similar approach for environment interaction, by providing the agent feedback on whether the command ran successfully, along with the output of running the command. For example, if the agent runs the generated test suite, we provide feedback on whether all tests passed (and the command succeeded or not), along with the terminal output containing any test error messages.

To encourage planning and reflection, we require TestForge to reflect on the output of any interaction with the environment. Self-reflection looks similar to chain-of-thought reasoning [55], where we show TestForge the command output and ask it to plan out subsequent steps systematically. For example, if the test suite generated by TestForge had three failing tests and failed to cover two lines, the agent is prompted to reflect, ideally producing a plan to fix the three failing tests and add a new test to cover the two missing lines.

```
Your goal is to improve the test suite at {test_file} to
     achieve **broad-coverage** of the code below.

...
IMPORTANT REQUIREMENTS:
1. Check coverage after each iteration
2. No external help or resources use only the snippet
     below.
...
Below is the **complete code snippet** to test:
{code_src}
...
Output the final test suite (20+ tests) for {test_file}
     in a single code block
```

Listing 1: TestForge prompt template

Listing 1 shows an abridged version of our agent prompt. We define the high level goal (generating a high coverage test suite) and provide the agent with a set of requirements for what a good test suite looks like and limitations of the environment (for example, no external dependencies). We prompt TestForge to generate greater than 20 tests, as this generally corresponds with higher mutation score and coverage.

```
 {
  "type": "function",
  "function": {
    "name": "execute_bash",
    "description": "Execute a bash command in the
         terminal...",
    "parameters": {
      "type": "object",
      "properties": {
         "command": {
           "type": "string",
            "description": "The bash command to
                 execute..."
         },
         ...
      },
      "required": [
        "command"
      ]
    }
  }
}
```

Listing 2: TestForge bash tool call JSON definition

```
{
    "id": "call_h3lCE3GtZaKOke9hqPCdv0Wy",
    "type": "function",
    "function": {
        "name": "execute_bash",
        "arguments": "{"command": "..."}"
    }
}
```

Listing 3: TestForge bash tool call LLM output

Listing 2 shows how we define both our agent actions and interactions with the environment in LiteLLM[4] as tools. This JSON definition is passed as input to the OpenAI API, which supports tool calling functionality. Listing 3 shows an example output of calling the `execute_bash` tool defined in Listing 2; the LLM supplies the required arguments and the name of the tool.

3.2 Agent Actions

We define the list of agent actions needed to navigate and edit complex code bases. We use the tool calling functionality of LiteLLM, which asks the LLM to generate a JSON object that represents a tool call. These tool calls are defined as a Python function, which performs the described functionality (for example viewing or editing a file). When the model outputs a well-formed JSON object representing the tool call, LiteLLM invokes the correct Python funtion. This is consistent with OpenHands's CodeAct agent [52], which achieves state-of-the-art performance on SWEBench [23].

3.2.1 Navigate repository. An important function for code agents is navigating *complex* code bases. We provide TestForge with the ability to search for files containing a particular prefix or suffix. We also instrument this with the ability to search across the entire repository or in a specific directory of the repository.

3.2.2 Write file. We provide TestForge the ability to create any file or overwrite any existing file. This is also in line with prior work [52, 57]. In practice, TestForge primarily writes Python test files. There is no limit on how long the generated file can be. This enables TestForge to generate tests in cases where the zero-shot test suite has many errors by overwriting the buggy test file.

3.2.3 Edit file. We provide editing functionality to TestForge for any file (also in line with prior work [52, 57]). TestForge primarily edits Python test files. The command takes in both original text and replacement text. The original text should occur once in the file. If the original text is not found in the file or occurs multiple times, the command fails, and the agent is prompted for another replacement. If the command succeeds, the changed lines are outputted to the agent, enabling iteration if the changes were not what was intended. We also provide functionality to insert lines into an existing file, enabling agents to add or insert new tests into an existing test file.

3.2.4 View file. Our view file tool works in tandem with our search functionality. An agent can view a range of 400 lines in a file (with the option to specify the range). Often TestForge wants to target a specific range, for example, a specific set of lines not covered by the existing test suite, which can be done by specifying the appropriate range. We choose 400 lines to enable TestForge to understand other files in the repository, while not overwhelming the model context with file content. This is consistent with both SWEAgent [57] and CodeAct [52].

[4]https://github.com/BerriAI/litellm

3.3 Environment Feedback

In addition to agentic actions, the system also provides feedback from the environment to TestForge. We define environment feedback in LiteLLM in the same way as agent actions. Here, we outline sources of feedback for TestForge when generating tests.

3.3.1 Test execution feedback. One important source of feedback is the output from test execution. We provide the agent with the command to execute the tests. Once tests are executed, we provide the full output from the execution of the tests, including any syntax or assertion errors in the generated test suite (assertion errors often contain expected output, making it easier for the agent to update the tests). Syntax errors can also be fixed by using the edit functionality. The output for the developer-provided tests fits in the 128k context of most LLMs, enabling us to include most test outputs in our prompt context.

3.3.2 Code coverage feedback. In addition to test execution feedback, we also provide the coverage report feedback. We provide the agent with the command to run the code coverage report. Our coverage report feedback includes the file, set of covered lines and a set of lines that are missing coverage. We use the coverage library in Python to measure line coverage and generate the coverage report. The agent can then use this information to view uncovered lines in the file under test and add tests that target these lines.

3.3.3 Bash command feedback. We also provide TestForge the ability to execute arbitrary `bash` commands and receive execution feedback from running them. We run all commands in a docker image with the dependencies for TestForge and the individual project to mitigate any associated safety risks. The output from this tool call includes whether the command succeeded or failed, and all terminal output (stdout and stderr).

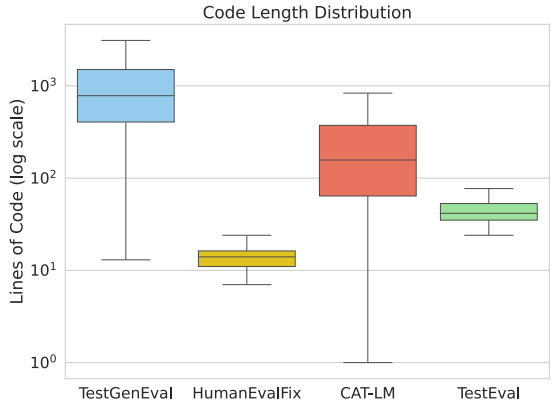
3.4 Implementation

TestForge is implemented in approximately 4.3k LOC of Python; our anonymous replication package is available at: https://anonymous.4open.science/r/OpenHands-7E28/. To replicate our results, see the README at `evaluation/benchmarks/testgeneval/README.md`.

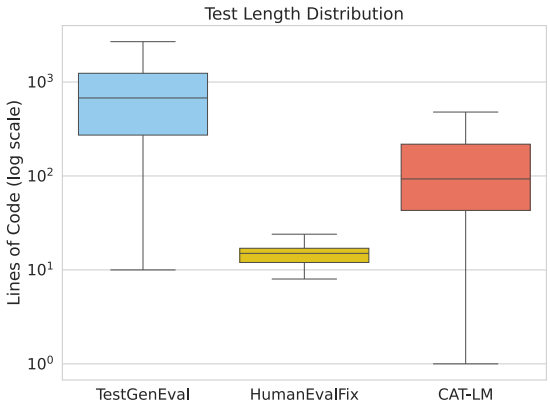
We integrate TestForge into OpenHands, a framework intended to support modular building and extension of software engineering agents, while facilitating reproducible AI agent research. We use GPT-4o as our model of choice, due to high performance on TestGenEval [22] and easy API access. However, because we use LiteLLM, it is easy to switch which model we use. We also adapt TestGenEval to work with any agent integrated with the OpenHands framework [52]. Our replication package includes the prompt for TestForge, code to run TestForge and the full agentic version of TestGenEval. We hope these contributions will enable others in the community to easily extend TestForge and evaluate future test generation agents.

4 Experimental Setup

We compare TestForge with Pynguin [30] and CodaMosa [25], current state-of-the-art unit test generation tools and GPT-4o 0-shot prompting on the TestGenEval benchmark [22]. We ask the following research questions:



(a) TestGenEval code lengths



(b) TestGenEval test lengths

Figure 3: Code and test lengths across TestGenEval, HumanEvalFix, CAT-LM, TestEval. Code and test files in TestGenEval are significantly longer than other benchmarks (even with the log scale). TestEval is not included in the test lengths plot, as it does not contain "gold" tests.

RQ1: Runtime Performance: How well does TestForge perform at generating high coverage test suites? We measure the pass@1, coverage and mutation score on a large scale benchmark of complex GitHub projects. We compare TestForge against Pynguin, CodaMosa, and zero-shot prompting, measuring the ability of each approach to generate full test suites.

RQ2: Lexical Performance: How similar are test suites generated by TestForge to actual developer test suites? We also measure lexical similarity of generated tests to developer written tests to understand whether tests generated by TestForge and other LLM approaches are more "human like" than search-based genetic programming approaches.

RQ3: Design Decisions: How does performance vary with number of iterations? Does performance improve by starting with the zero-shot solution? We measure the effect of varying the maximum number of iterations of agentic feedback on coverage. We also examine the insight that starting with the zero-shot solution

is better than starting from scratch by performing an ablation of coverage and mutation score between these two approaches.

RQ4: Behavior: Which actions does TestForge take most frequently? How does the readability and maintainability of tests generated by TestForge compare against other baselines? We perform both a quantitative analysis of actions taken by TestForge and a small case study of tests generated by TestForge and each of the baselines. We discuss the readability and maintainability of test suites for each approach.

4.1 Dataset

We evaluated all baselines and TestForge on TestGenEval [22], a benchmark of code test file pairs, sourced from 11 large-scale open source repositories (3,523-78,287 stars). Figure 3 shows the distribution of code and test lengths across TestGenEval, HumanEvalFix, CAT-LM, and TestEval. Code and test files are much longer in TestGenEval than other benchmarks: code files are on average 1157 LOC and test files are on average 943 LOC, while for CAT-LM, the benchmark with the second-longest files, code files are on average 344 LOC and test files are on average 211 LOC. This simulates a much more realistic experience of writing tests for complex repositories.

We choose TestGenEval over existing benchmarks such as CAT-LM [43], TestEval [51] and HumanEvalFix [10], as these existing benchmarks primarily target smaller code and test files, simple problems such as LeetCode and small programming problems. As a result, even zero-shot approaches saturate these benchmarks (coverage values greater than 85% for GPT-4o) [47, 51]. TestGenEval provides a benchmark that existing models perform poorly on, where we can measure the impact of execution feedback.

4.2 Baselines

We compare TestForge against multiple classical and LLM baselines. We run TestForge for 25 iterations, and classical baselines for 600 seconds (the average time per data point of TestForge is 447 seconds). For classical baselines, we choose Pynguin [30] and CodaMosa [25]. Pynguin uses genetic programming to search the input space and maximize code coverage over the code under test, monitoring the output of the code under test with each generated input. These input and output pairs are then converted into test cases. CodaMosa [25] extends this by using LLMs when Pynguin hits a coverage plateau (the same coverage for 25 iterations). We upgrade the model used in CodaMosa from Codex [29] to GPT-4o [37] (Codex is no longer available in the OpenAI API).

Both Pynguin and CodaMosa are only listed as compatible with Python 3.10. Of the 1210 programs in TestGenEval, only 45 use Python 3.10. We compare against both baselines on this subset of data. Furthermore, both baselines do not successfully generate tests for all 45 programs; CodaMosa only generates tests for 28 programs, and Pynguin generates tests for a different 27 programs. Errors include segmentation faults with both tools or issues with dependency analysis for complex projects (we raised an issue but the fix is not simple).[5] For programs where either baseline fails to generate a test, we mark pass@1, coverage and mutation score as 0.

In addition to these classical baselines, we compare against two LLM baselines: GPT-4o 0-shot [36] and CAT-LM [43]. GPT-4o 0-shot provides a baseline for agentic improvement, as we start with 0-shot tests with TestForge. We use the `gpt-4o-2024-08-06` version of GPT-4o. CAT-LM is a state-of-the-art LLM for test generation, trained on an aligned data set of code and test files. By pretraining with this aligned set, CAT-LM is able to outperform much larger models with larger pretraining budgets. For both LLM baselines we follow the original TestGenEval paper [22], using the same prompt and temperature of 0.2. Since both LLM baselines work on the full TestGenEval benchmark, we measure performance both on the entire 1210 programs and the subset of 45 programs that are compatible with Pynguin and CodaMosa.

Other LLM baselines exist as well, including MuTAP [15], HITS [53] and CoverUp [41], however we were not able to compare against them. Unfortunately, MuTAP is exclusively integrated with HumanEvalFix, and cannot take arbitrary context; we therefore cannot trivially evaluate it on TestGenEval. HITS targets Java 17 and is not is not compatible with Python. Both HITS and CoverUp struggle with long context methods; these approaches iteratively refine coverage for a specific focal method, which does not scale to large files such as those in TestGenEval with a large number of methods. CoverUp costs approximately $2 per data point in TestGenEval, due to operating at the method level. Even with TestForge that costs $0.63 cents per file our experiments cost $762 for all of TestGenEval; running CoverUp would cost $2400, which is out of our price range.

4.3 Metrics

We measure test adequacy with both runtime and lexical metrics. Runtime metrics approximate the quality of generated test suite, while lexical metrics measure similarity between generated test suites and developer-written test suites. We report the cost of TestForge in USD to ensure that our approach is economically viable.

4.3.1 Runtime Metrics. **Pass@1** measures whether the generated test suite has *any* test that passes for the code under test. We can add the resulting test suite to the existing code base by removing all failing tests. High pass@1 indicates that generated tests can be added to a test suite, but does not provide any guarantee of the quality of generated tests.

Coverage measures the proportion of code lines executed by passing tests in generated test suite. We omit failing tests from the coverage computation; these tests would require developer modification before being added to an existing test suite. Coverage serves as a weak proxy for test quality; high coverage indicates a test suite that executes most of the code under test, but does not guarantee the written tests have meaningful assertions.

Mutation score measures the percentage of synthetic bugs injected detected by the test suite (the test suite should pass on the original code and fail on the buggy code). To compute mutation score, we introduce synthetic bugs into the code under test and measure if the tests can detect these bugs. We use cosmic ray[6] as our mutation testing tool and use the standard set of operators. We also set a one hour timeout in line with TestGenEval (which only has a 1.06% error in mutation score results). Mutation score provides a more robust measure of test suite quality than other metrics [24, 35]. High mutation score indicates a test suite is capable of catching future

[5] Issue elided for double blind purposes

[6] https://github.com/sixty-north/cosmic-ray

Model	Pass@1	Coverage	Mutation Score
Full TestGenEval (1210 Programs)			
GPT-4o (0-shot)	64.0%	34.8%	18.4%
TestForge	**84.3%**	**44.4%**	**33.8%**
Python 3.10 TestGenEval (45 Programs)			
GPT-4o (0-shot)	97.8%	54.0%	27.3%
Pynguin	60.0%	24.0%	4.2%
CodaMosa	62.2%	30.2%	2.7%
TestForge	**100.0%**	**60.0%**	**31.6%**

Table 1: Runtime results on the full TestGenEval dataset and Python 3.10 subset. TestForge achieves higher pass@1, coverage and mutation score than all baselines. All differences between baselines and TestForge are statistically significant ($p < 0.05$) other than GPT-4o 0-shot ($p = 0.07$) for the Python 3.10 subset.

Model	CodeBLEU	ROUGE	Edit Sim
Full TestGenEval (1210 Programs)			
CAT-LM	29.0	6.8	25.2
GPT-4o (0-shot)	31.8	22.9	25.9
TestForge	**32.1**	**23.0**	**27.0**
Python 3.10 TestGenEval (45 Programs)			
CAT-LM	29.6	4.8	21.8
GPT-4o (0-shot)	**35.0**	28.6	25.8
Pynguin	18.2	9.7	14.2
CodaMosa	11.3	10.0	17.5
TestForge	33.3	**29.3**	**26.9**

Table 2: TestGenEval lexical results for all 1210 programs in the dataset. All LLM-based approaches achieve similar scores on all lexical metrics (besides the low ROUGE score of CAT-LM), but genetic programming approaches achieve far lower lexical performance.

bugs that may be introduced into the code under test, and thus is relatively robust. However, mutation score is costly to compute, as we have to run the entire test suite for each bug.

4.3.2 Lexical Metrics. **CodeBLEU** [44] serves as a proxy of similarity between code snippets. Based on BLEU score, CodeBLEU considers n-gram match between both code pieces. It also incorporates code specific similarity measures such as the BLEU score of reserved keywords, AST match, and data flow match.

ROUGE [27] measures the longest overlapping subsequence of tokens between the generated test and gold test, using F1 score. ROUGE has been used in prior testing work [32, 43] as a metric of code similarity; high ROUGE indicates substantial overlap between developer written tests and generated tests.

Edit Similarity is the single character similarity between the generated test suite and gold standard developer-provided test suite for a file under test. Specifically it is 1 - Levenstein edit distance – number of single character edits needed to transform the generated test to the gold test, normalized by the total number of characters. High edit similarity indicates similar tests to developer written tests, while low edit similarity indicates different tests.

5 Results and Analysis

We report results for each research question and discuss their implications compared to other test generation approaches.

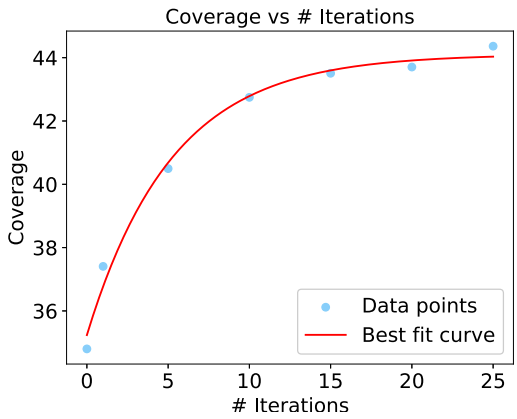
5.1 RQ1: Runtime Performance

Table 1 shows the pass@1, coverage and mutation score of the test suites generated by GPT-4o and TestForge on all 1210 programs in TestGenEval. Pass@1 is relatively high for TestForge, with TestForge generating tests for 84.3% of all programs. 0-shot prompting does significantly worse, only generating passing tests for 64.0% of all programs. Evaluations of other previous techniques on benchmarks like HumanEvalFix or TestEval tend to produce higher coverage results; we note the relative complexity of the code in TestGenEval compared to these other benchmarks. That said, TestForge outperforms 0-shot prompting by 9.6%. This indicates that even for complex test suites, models still benefit from both execution feedback and coverage reports detailing missing lines. The generally low coverage can be attributed to the long code files present in TestGenEval, which are frequently 10,000+ tokens. Mutation score varies more between both approaches, with TestForge outperforming 0-shot prompting by 15.0%. This indicates that using an agentic approach not only results in more code paths being executed in the code under test, but also in higher quality tests for the covered code. CAT-LM performs very poorly with 0% pass@1, coverage and mutation score. CAT-LM is a very small LLM (approximately 3B parameters), therefore does not have the same test generation capabilities of larger models. All observed differences in results are statistically significant ($p < 0.05$).

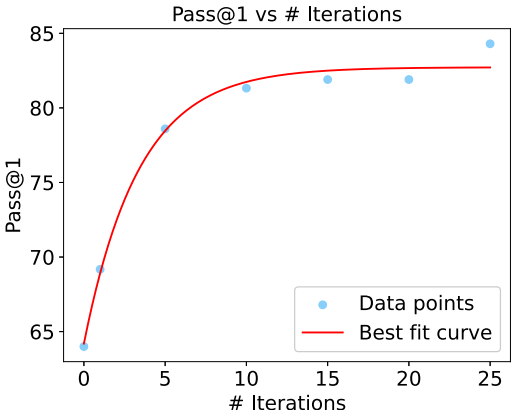
We also report runtime metrics in comparison against CodaMosa and Pynguin on the 45 data point subset that uses Python 3.10 (a requirement for both CodaMosa and Pynguin). For cases where Pynguin and CodaMosa do not generate tests, we mark both the coverage and mutation score of the respective approach as 0%. CAT-LM performs very poorly on this subset as well with 0% pass@1, coverage and mutation score. For this subset of TestGenEval, pass@1 tends to be relatively high with both 0-shot and TestForge achieving nearly 100% pass@1.

Pynguin and CodaMosa struggle more, not generating tests for all cases, largely due to issues with analyzing dependencies for these complex programs. Coverage also varies significantly between approaches; Pynguin and CodaMosa struggle to generate high coverage test suites for these complex programs (even with the suggested 600 second budget for test generation in CodaMosa [25]). Mutation score is even lower for baseline approaches, with genetic programming approaches primarily optimizing for high code coverage rather than mutation score. As a result, TestForge generates higher coverage test suites both genetic programming approaches by greater than 10% and 6% when compared to 0-shot. The difference for mutation score is even greater with a greater than 25% difference between TestForge and both search-based baselines.

Even excluding cases where Pynguin and CodaMosa fail to generate tests, TestForge outperforms both approaches with a coverage



(a) Coverage with TestForge iterations



(b) Pass@1 with TestForge iterations

Figure 4: Coverage and pass@1 in comparison to number of iterations. Both metrics have diminishing returns as k, increases, indicating k=25 is a good value.

difference of 20.9% and 12.2% respectively and a mutation score difference of 23.2% and 30.8% respectively. All observed performance differences in results are statistically significant other than between TestForge and 0-shot, where the p-value is 0.07.

Finally, TestForge generates test suites in its default configuration at an average cost of $0.63 per file under test, which we argue is reasonable efficient. We evaluate the impact of running fewer iterations (providing potential cost savings) in Section 5.3.

5.2 RQ2: Lexical Performance

We report lexical performance as a measure of how similar generated test suites are to developer written test suites. Table 2 shows the CodeBLEU, ROUGE and edit-similarity of all LLM approaches. Lexical scores are generally low for TestGenEval; the "gold standard" developer-written test files are relatively long, comparatively, and existing models struggle to generate long test suites that are comprehensive. However, the generated tests are relatively natural and similar to developer-written tests, especially when compared to search-based test generation approaches. LLMs are pretrained on human code, whereas search-based approaches have no notion of "naturalness".

Model	Pass@1	Coverage
TestForge (no seeding)	79.0%	42.1%
TestForge	**84.3%**	**44.4%**

Table 3: TestGenEval runtime results on all 1210 programs in the dataset, with and without 0-shot seeding. Removing the 0-shot seeding results in a 2.3% drop in coverage.

We also report lexical metrics for both LLM and search-based approaches on the Python 3.10 subset of TestGenEval. LLM approaches have CodeBLEU, ROUGE and edit-similarity of approximately 30%, while search based approaches perform worse on all metrics by greater than 9%. CodaMosa and Pynguin also perform poorly across all metrics; in general tests generated by these approaches use poor variable names and inputs that are more complex than typically present in human-written tests. TestForge and 0-shot slightly outperform CAT-LM, but the differences are relatively minor, with little to no difference between 0-shot and TestForge.

5.3 RQ3: Design Decisions

We next evaluate the impact of design decisions involved in TestForge. Specifically, we look at the impact of (1) the number of agent iterations on test suite quality, and (2) providing the zero-shot test file as a starting point for TestForge. We evaluate these in terms of coverage and pass@1 scores for the resulting test suites.

5.3.1 Number of iterations. We measure the performance as a function of the number of iterations of TestForge. The average cost per iteration is only four cents, making our approach relatively low-cost even as we scale up the number of iterations. More iterations provide more opportunities for TestForge to iterate on execution feedback and target lines that lack coverage.

Figure 4 shows both coverage and pass@1 as we increase the number of iterations. Coverage increases significantly in the first 10 iterations (almost 10%), but after these 10 iterations the improvement in coverage is much more incremental. In a cost-constrained setting, one could use TestForge with only 10 iterations, halving the cost with only minimal performance loss. Pass@1 follows a similar trend, with significant gains in the first five iterations and only incremental improvement afterwards. The curve for pass@1 is steeper than coverage because the criteria is less fine-grained (if only one generated test passes in the test suite, pass@1 is 1).

5.3.2 Removing the zero-shot starting test file. Another key insight behind TestForge is starting from the 0-shot generated test suite rather than from scratch. Intuitively, this allows the model to further improve coverage in cases where 0-shot prompting produces a reasonable test suite, while overwriting the existing test file in cases where the 0-shot generated test suite is far from correct. To make the comparison fair, we provide TestForge without seeding with an additional iteration.

Table 3 shows the coverage and pass@1 of TestForge with and without zero-shot seeding. Both pass@1 and coverage go down without the seeding of the 0-shot file, with a 5.3% drop in pass@1 and a 2.3% drop in code coverage. 0-shot solutions are often not far from correct and can easily be iterated on to generate high quality test suites.

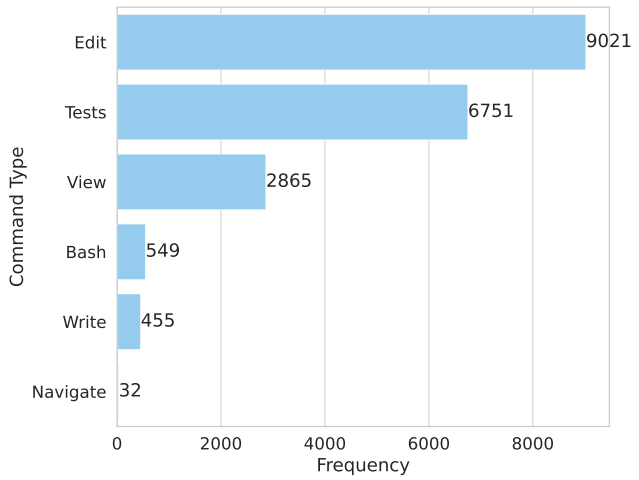


Figure 5: Frequency of TestForge commands taken while generating tests for all 1210 programs in TestGenEval. The most common actions are editing and executing the generated test suite, indicating an iterative approach to test suite refinement.

5.4 RQ4: Behavior

We perform both a quantitative analysis of actions taken by TestForge (Section 5.4.1) and a qualitative analysis of tests generated by TestForge and other baselines (Section 5.4.2).

5.4.1 Quantitative Action Analysis. Figure 5 shows the full set of actions taken by TestForge while generating tests for all 1210 programs in TestGenEval. "View" corresponds to viewing files, while edit, navigate and write correspond to editing the contents of a file, navigating the repository and writing a new file. "Tests" correspond to executing the generated test suite and obtaining both test and coverage feedback. Finally, "Bash" refers to executing arbitrary bash commands. The most frequent actions taken by TestForge are editing and executing the test suite (and also obtaining coverage information). We hypothesize the large number of edits and test execution corresponds to TestForge iterating on execution feedback (similar to what we see in Figure 1). Navigation is much less frequently used, suggesting that the complex dependency analysis used by prior work [41, 53] might not be as important as code editing. In cases, where TestForge used navigation functionality, it was in the first few iterations, and dominated by code edits and test execution later. The most frequent bash scripts we observed were invocations to the Python interpreter (e.g. `python -c ...`), indicating TestForge uses our bash functionality to understand the execution behavior of the code under test.

5.4.2 Qualitative Readability Analysis. Following prior work [43], we randomly select three programs in TestGenEval where all baselines successfully generate tests with coverage. We highlight an example of tests generated by TestForge, and all baselines, discussing the trade-offs from a perspective of readability and maintainability. The other two examples can be found in Supplementary Materials.

```
@pytest.mark.xfail(strict=True)
def test_case_0():
    var_0 = module_0.__dir__()
    module_1.convert_label_indexer(var_0, var_0,
        tolerance=var_0)
```

Listing 4: Pynguin test

```
def test_case_2():
    tuple_0 = ()
    vectorized_indexer_0 =
        module_0.VectorizedIndexer(tuple_0)
    assert vectorized_indexer_0 is not None
    assert module_0.dask_array_type == ()
    assert len(module_0.integer_types) == 2
```

Listing 5: CodaMosa test

```
def test_convert_label_indexer():
    index = pd.Index([1, 2, 3])
    assert indexing.convert_label_indexer(index, 2)
        == (1, None)
    assert indexing.convert_label_indexer(index,
        slice(1, 3)) == (slice(1, 3), None)
```

Listing 6: GPT-4o test

```
def test_convert_label_indexer():
    index = pd.Index([1, 2, 3])
    assert indexing.convert_label_indexer(index, 2)
        == (1, None)
    assert indexing.convert_label_indexer(index,
        slice(1, 3)) == (slice(0, 3), None)
```

Listing 7: TestForge (repaired) test

Figure 6: Tests generated for the `label_indexer` method. Pynguin and CodaMosa generate tests that are very hard to maintain (poor variable names, unintuitive values). GPT-4o generates a good test, with a subtle bug that TestForge fixes.

Pynguin: Pynguin generates a test that covers the method under test. However, the test contains many attributes that hinder maintainability. The test is named poorly (`test_case_0`) and variables are named unintuitively. The test is also marked as expected fail, when it isn't clear what error or exception is being tested. Even the imports are named poorly (with the module being called `module_0`). This is an inherent limitation of search based approaches; since they are not trained to mimic developer tests, the generated tests tend to look very different (also seen with low lexical metric scores).
CodaMosa: CodaMosa suffers from many of the same issues as Pynguin, because it is built off Pynguin. CodaMosa prompts LLMs to escape coverage plateaus in Pyguin's genetic search based approach. However, despite prompting LLMs, CodaMosa converts the variable names and inputs to the format that Pynguin uses. In this case, CodaMosa fails to generate a test directly covering the method under test. The other test shown suffers from the same problems as Pynguin (poor variable names, poor test name, importing the module as `module_0`).
GPT-4o 0-shot: Unlike other baselines, GPT-4o generates a well-structured test. The GPT-4o generated test uses appropriate test and variable names. The inputs and expected outputs are also well formatted and easy to understand, with module imports using the correct names (pd for pandas). However, unfortunately the test

does not pass due to a subtle bug in the expected output (should be `slice(0, 3)` instead of `slice(1, 3)`).

TestForge: TestForge generates a passing test that is both readable and easy to maintain. GPT-4o provides a good starting point for test generation, with a relatively good test structure and readable assert statements. By fixing the bug in the zero-shot generation TestForge can improve overall test suite coverage, while also producing an easy to maintain test that can be added to a developer test suite.

6 Limitations

We outline potential limitations with TestForge and discuss their impact on our reported results.

Data contamination: A limitation of TestForge is that GPT-4o might have seen the TestGenEval test set at pretraining time. However, currently this does not seem to be a major issue, as even state-of-the-art agents still achieve relatively low performance on TestGenEval. Furthermore, we measure performance improvements of TestForge in comparison to the base model, making data leakage less of a concern, as we are concerned about relative performance differences rather than absolute performance values. The larger and newer models also have lower data contamination rates due to the large number of tokens present at the pretraining time [42].

Generalization of findings: Another limitation is that our findings might not generalize to all repositories. We specifically target Python repositories with TestForge, and benchmark on TestGenEval, which is adapted from the SWEBench dataset. While the code and test files in TestGenEval are sourced from complex GitHub projects, the repositories in TestGenEval are widely popular. This might make them easier to test than other domain-specific or company specific repositories.

Oracle problem: One other limitation of our approach (and most automatic test generation approaches) is the oracle problem. One assumption behind TestForge is that the code under test is correct. However this might not be the case, as generated tests may fail on the code under test, while exposing bugs in the code under test. Despite this, our approach is still useful in catching regressions or bugs introduced in future versions of the code under test.

7 Related Work

Classical Test Generation: Classical test generation techniques employ both black box and white box techniques to find "breaking" inputs for the code under test. Random/fuzzing techniques such as Randoop [39], aflplusplus [17] and honggfuzz maximize coverage by preserving inputs that improve coverage. Property testing tools such as Korat [7], QuickCheck [13] and Hypothesis [31] enable the systematic checking of user-defined properties. PeX [48] and Eclipser [12] use symbolic execution to mathmatically reason over a large input space and produce crashing input. The issue with both fuzzing and symbolic execution is they rely on the program crashing to find failing inputs, limiting the extend they can test programs. Search based test generation approaches [18, 25, 30] solve this problem by assuming the code under test is correct. These approaches maximize coverage by exploring test inputs that improve coverage and creating a large test suite of these inputs. However, search-based approaches struggle to generate "natural" tests, suffering from both stylistic and readability problems [8, 14, 45].

Neural Test Generation: More recently, neural test generation methods leverage language models to generate readable and natural tests. ConTest [50] employs a generic transformer model that utilizes the tree representation of code to generate assert statements. Building on this, several approaches [16, 21, 49, 54, 56] leverage transformer architectures for assertion generation. Their results demonstrate that LLM generated assertions are more natural and preferred by developers compared to existing tools such as EvoSuite. TeCo [32] further broadens the scope of the completion of the test by generating test statements one at a time. CAT-LM [43] leverages a LLM pretrained on aligned code and test filepairs to generate high quality unit tests. While these neural approaches address many readability challenges inherent in classical test generation methods, they primarily focus on generating individual tests, offering significantly fewer time-saving benefits.

Large Language Models of Code: Large language models (LLMs) can perform well across many software engineering tasks [9, 20, 33, 34]. TestPilot [46] uses GitHub's Copilot to generate unit tests. GPT-4o is a state of the art LLM, with top performance on many benchmarks [22, 29, 37]. Empirical studies have evaluated LLMs efficacy in code and test generation, as well finding that performance varies heavily between benchmark and code base [46, 58]

Test Generation Agents: Recently, there has been extensive work on developing software engineering and software testing agents. SWEAgent [57] leverages execution feedback to better solve software engineering pull requests. ChatUniTest [11] and MuTAP [15] target focal method generation, using coverage and mutation score as feedback. Unfortunately, due to targeting each method individually, both methods do not scale well to large repositories, becoming costly to use (average code file in TestGenEval contains 58 focal methods, with each method requiring many iterations with the agent). CoverUp [41] and HITS [53] extend the prior work by adding dependency analysis and error report feedback, but still suffer from cost issues due to their method level approach. TestForge takes a more fluid approach, enabling the LLM to view dependencies as it sees fit, while also allowing it to generate multiple tests at a time.

8 Conclusion

In this work, we introduced TestForge, an agentic, feedback-driven test generation framework that iteratively refines test suites based on execution results and coverage reports. By leveraging an agentic loop, TestForge effectively balances test readability, coverage, and cost efficiency, outperforming both search-based and LLM-based baselines. Our experiments on TestGenEval demonstrate a significant improvement in pass@1 (84.3%), code coverage (44.4%), and mutation score (33.8%), while maintaining cost-efficiency ($0.63 per test file). The produced tests are also syntactically more similar to human-produced tests than prior classic automated approaches. The agentic feedback loop enables dynamic adaptation, refining tests iteratively, to address both coverage gaps and mutation score deficiencies. Our ablation studies further highlight the benefits of starting with zero-shot prompting and iterating on execution feedback, key design decisions. By integrating with OpenHands, we hope to encourage future advancements in test generation research and foster more effective and scalable automated testing solutions.[7]

[7] Anonymized replication: https://anonymous.4open.science/ r/OpenHands-7E28/

9 Data Availability

We release both the agentic version of TestGenEval and the prompt and implementation of TestForge at https://anonymous.4open.science/r/OpenHands-7E28/. The README detailing how to reproduce all of our experiments is available at `evaluation/benchmarks/testgeneval/README.md`.

References

[1] 2021. GitHub Copilot. https://github.com/features/copilot

[2] 2025. Cursor. https://www.cursor.com/en

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Computing Survey* 51, 3 (2018), 50–88.

[4] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient Training of Language Models to Fill in the Middle. *arXiv:abs/2207.14255* (2022).

[5] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, How, and Why Developers (Do Not) Test in Their IDEs. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE '15)*. 179–190.

[6] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2015. How (Much) Do Developers Test?. In *International Conference on Software Engineering* (Florence, Italy) *(ICSE '15)*. 559–562.

[7] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. *SIGSOFT Software Engineering Notes* 27, 4 (2002), 123–133.

[8] Carolin Brandt and Andy Zaidman. 2022. Developer-Centric Test Amplification: The Interplay between Automatic Generation Human Exploration. *Empirical Software Engineering* 27, 4 (2022), 35 pages.

[9] Tom B. Brown et al. 2020. Language Models Are Few-Shot Learners. In *NeurIPS*.

[10] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv:abs/2107.03374* (2021).

[11] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. arXiv:2305.04764 [cs.SE] https://arxiv.org/abs/2305.04764

[12] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-Box Concolic Testing on Binary Code. In *International Conference on Software Engineering (ICSE '19)*. 736–747.

[13] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference on Functional Programming (ICFP '00)*. 268–279.

[14] Ermira Daka, José Miguel Rojas, and Gordon Fraser. 2017. Generating Unit Tests with Descriptive Names or: Would You Name Your Children Thing1 and Thing2?. In *International Symposium on Software Testing and Analysis (ISSTA '17)*. 57–67.

[15] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. 2023. Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing. arXiv:2308.16557 [cs.SE] https://arxiv.org/abs/2308.16557

[16] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *International Conference on Software Engineering (ICSE '22)*. 2130–2141.

[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Conference on Offensive Technologies (WOOT '20)*. 10–10.

[18] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. 416–419.

[19] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *ICLR*.

[20] Aaron Grattafiori et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[21] Soneya Binta Hossain and Matthew Dwyer. 2024. TOGLL: Correct and Strong Test Oracle Generation with LLMs. arXiv:2405.03786 [cs.SE] https://arxiv.org/abs/2405.03786

[22] Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. 2024. TestGenEval: A Real World Unit Test Generation and Test Completion Benchmark. arXiv:2410.00752 [cs.SE] https://arxiv.org/abs/2410.00752

[23] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[24] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 654–665. doi:10.1145/2635868.2635929

[25] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 919–931. doi:10.1109/ICSE48619.2023.00085

[26] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (<conf-loc>, <city>Lisbon</city>, <country>Portugal</country>, </conf-loc>) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 52, 13 pages. doi:10.1145/3597503.3608128

[27] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Conference on Text Summarization Branches Out*. 74–81.

[28] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. 2024. Exploring and Evaluating Hallucinations in LLM-Powered Code Generation. arXiv:2404.00971 [cs.SE]

[29] Shuai Lu et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *NeurIPS Datasets and Benchmarks*.

[30] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: automated unit test generation for Python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 168–172. doi:10.1145/3510454.3516829

[31] David MacIver, Zac Hatfield-Dodds, and Many Contributors. 2019. Hypothesis: A New Approach to Property-Based Testing. *Journal of Open Source Software* 4, 43 (2019), 1891.

[32] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. In *International Conference on Software Engineering (ICSE '23)*. 2111–2123.

[33] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *arXiv:abs/2305.02309* (2023).

[34] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *ICLR*.

[35] A Jeeerson Ooutt and Jeerey M Voas. 1996. Subsumption of Condition Coverage Techniques by Mutation Testing. https://api.semanticscholar.org/CorpusID:9492620

[36] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[37] OpenAI et al. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[38] Wendkûuni C. Ouédraogo, Kader Kaboré, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. 2024. Large-scale, Independent and Comprehensive study of the power of LLMs for test case generation. arXiv:2407.00225 [cs.SE] https://arxiv.org/abs/2407.00225

[39] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Conference on Object-Oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. 815–816.

[40] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. 2020. Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities. In *International Conference on Software Maintenance and Evolution*. 523–533.

[41] Juan Altmayer Pizzorno and Emery D. Berger. 2025. CoverUp: Coverage-Guided LLM-Based Test Generation. arXiv:2403.16218 [cs.SE] https://arxiv.org/abs/2403.16218

[42] Daniel Ramos, Claudia Mamede, Kush Jain, Paulo Canelas, Catarina Gamboa, and Claire Le Goues. 2024. Are Large Language Models Memorizing Bug Benchmarks? arXiv:2411.13323 [cs.SE] https://arxiv.org/abs/2411.13323

[43] Nikitha Rao, Kush Jain, U. Alon, Claire Le Goues, and Vincent Joshua Hellendoorn. 2023. CAT-LM Training Language Models on Aligned Code And Tests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 409–420. doi:10.1109/ASE56229.2023.00193

[44] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE] https://arxiv.org/abs/2009.10297

[45] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. 2011. Scaling Up Automated Test Generation: Automatically Generating Maintainable Regression Unit Tests for Programs. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ASE '11)*. 23–32.

[46] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. arXiv:2302.06527 [cs.SE] https://arxiv.org/abs/2302.06527

[47] Yuling Shi, Songsong Wang, Chengcheng Wan, and Xiaodong Gu. 2024. From Code to Correctness: Closing the Last Mile of Code Generation with Hierarchical Debugging. arXiv:2410.01215 [cs.CL] https://arxiv.org/abs/2410.01215

[48] Nikolai Tillmann and Peli de Halleux. 2008. Pex - White Box Test Generation for .NET. In *Tests and Proofs (TAP '08, Vol. 4966)*. 134–153.

[49] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit Test Case Generation with Transformers. *CoRR* abs/2009.05617 (2020).

[50] Johannes Villmow, Jonas Depoix, and Adrian Ulges. 2021. ConTest: A Unit Test Completion Benchmark featuring Context. In *Workshop on Natural Language Processing for Programming*. 17–25.

[51] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2024. TESTEVAL: Benchmarking Large Language Models for Test Case Generation. arXiv:2406.04531 [cs.SE] https://arxiv.org/abs/2406.04531

[52] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. arXiv:2402.01030 [cs.CL] https://arxiv.org/abs/2402.01030

[53] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. arXiv:2408.11324 [cs.SE] https://arxiv.org/abs/2408.11324

[54] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On Learning Meaningful Assert Statements for Unit Test Cases. *CoRR* abs/2002.05800 (2020).

[55] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1800, 14 pages.

[56] Robert White and Jens Krinke. 2020. ReAssert: Deep Learning for Assert Generation. *CoRR* abs/2011.09784 (2020).

[57] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793 [cs.SE] https://arxiv.org/abs/2405.15793

[58] Lin Yang et al. 2024. On the Evaluation of Large Language Models in Unit Test Generation. arXiv:2406.18181 [cs.SE] https://arxiv.org/abs/2406.18181