Exploiting Test Structure to Enhance Language Models for Software Testing

Kush Jain

CMU-S3D-25-100

April 8, 2025

Software and Societal Systems
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Claire Le Goues, Chair, Carnegie Mellon University
Christian Kaestner, Carnegie Mellon University
Daniel Fried, Carnegie Mellon University
Alex Groce, Northern Arizona University

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This material is based upon work supported in part by Gramma Tech (award 5002552), the National Science Foundation (awards CCF1910067 and CCF2129388), Microsoft (award 5008587), and Internally Funded Projects (award 5007039).

Keywords: software testing; machine learning; large language models

Dedicated to my grandfather, and greatest supporter, Surinder Kumar Jain, in loving memory.

Abstract

Software testing is an integral part of software development. However, testing faces challenges due to the time-consuming and challenging nature of writing high-quality tests, leading to poorly maintained test suites and lower overall software quality. Prior work for automatically generating tests, like EvoSuite and Randoop can generate high-coverage tests, however, often these tests are hard to read, unrealistic, or incorrect, necessitating additional effort from developers for verification. In contrast, language models have shown promise in generating human-like, high-quality code functions, benefiting tools like Copilot in code generation.

However, language models are not as successful at generating tests, struggling with both hallucination and with correctly invoking internal methods present in the code under test. This is because code generation language models are typically trained primarily for code generation and code completion. Benchmarks also do not resemble real-world development; existing benchmarks consist of simple programming or LeetCode problems. To help overcome these limitations, I focus on how we can incorporate domain-specific properties of testing such as the strong coupling between source and test files along with important test execution data to improve the evaluation and application of language models to software testing. I also examine how we can better evaluate test generation approaches with metrics that are more meaningful to developers and evaluation on larger codebases that more closely resemble real-world development. My thesis statement is: We exploit the structure of test code and close relationship between code and test files to improve the evaluation and application of language models to software testing in both pretraining and fine-tuning. This insight can (a) generate useful unit test cases, (b) identify weaknesses in existing test suites, (c) build more realistic test generation benchmarks, and (d) generate test suites for large scale projects.

My thesis will make the following contributions:

1. It presents a new method for pretraining models for test generation, that considers the relationship between source code and test code.
2. It provides an approach to automatically classify mutants as detected or undetected without executing the test suite by leveraging additional *test* context.
3. It evaluates all provided techniques with metrics and experiments that are practically meaningful to developers, not considered in prior work.
4. It introduces a benchmark for evaluating test generation approaches that is sourced from large scale open source repositories and thus more closely resembles real-world test generation.
5. It demonstrates the effectiveness of adding execution context to test generation models, which enables us to generate high quality test suites for large scale projects.

My work (ASE 2023) demonstrated that pretraining language models on dual objectives of code and test generation significantly improves unit test generation. I also leveraged the joint relationship between code and tests (FSE 2023) to improve predictive mutation testing techniques, modeling mutants at the token level, and in-

corporating both source and test methods during fine-tuning. I improved test generation evaluation (ICLR 2025) by introducing a large test generation benchmark, TestGenEval, that is sourced from large scale open source repositories. Finally, I built a test generation agent (submitted to ICSE 2026) that incorporates execution feedback, while also scaling to the large open source repositories in TestGenEval.

Acknowledgments

My PhD journey has been both a challenging and rewarding experience that I will always remember dearly. Reflecting on the experience, there were so many amazing people who lifted me up even in the hardest of times.

First and foremost, I would like to thank my advisor Claire Le Goues for her unwavering support, guidance, and mentorship throughout my PhD. Claire has been an incredible advisor, always pushing me to do my best work, and providing me with the resources and support I needed to succeed. I am grateful for her patience, kindness, and wisdom, and I am so thankful to have had the opportunity to work with her. I will always miss our weekly meetings where I would learn so much about research, life, and everything in between and hope that we can continue to collaborate in the future.

Reflecting on my undergraduate years, I would also like to thank Milos Gligoric for his guidance and mentorship. Milos has been an amazing mentor; I would not be the same researcher and be where I am currently without all of his support and guidance. I am so grateful for his patience, kindness, and wisdom, and I am so thankful for the opportunity to work with him. I would also like to thank his students who played a formative role in my development as a researcher: Ahmet Celik, Jiyang Zhang, Karl Palmskog, Kayvan Mansoorshahi, Marinela Parovic, Nader Al Awar, Pengyu Nie and Yu Yuki Liu.

I would like to thank my committee members, Alex Groce, Christian Kaestner, and Daniel Fried for their valuable feedback and support throughout my PhD. I am grateful for their time and effort in helping me improve my work, and I am so thankful for their guidance and mentorship. Their guidance has truly shaped my research and my career, and I am so grateful for their support.

I would like to thank my collaborators Baptiste Roziere, Gabriel Synnaeve, Kiran Kate, Martin Hirzel, Uri Alon, and Vincent Hellendoorn for their invaluable contributions to my research. I am so grateful for their insights, feedback, and support, and I am so thankful for the opportunity to work with them. I am so grateful for their guidance and mentorship, and I am so thankful for their support.

I would like to thank my lab mates: Aidan Yang, Claudia Mamede, Daniel Ramos, Harrison Green, He Ye, Jeremy Lacomis, Kaia Newman, Luke Dramko, Nikitha Rao, Paulo Santos, Sophia Kolak, Tobias Dürschmid, and Trenton Tabor. I am so grateful for their kindness, encouragement, and support, and will always remember our lab meetings where I developed so much as a researcher and a speaker. I am very grateful to have had the opportunity to work with such a talented and supportive group of people, and I am so thankful for their friendship and mentorship.

My PhD would not be the same without such loving and supportive friends. I would like to thank my amazing CMU friends: Aditi Kabra, Aidan Yang, Akhil Padmanabha, Carolina Carreira, Catarina Gamboa, Chenyang Yang, Claudia Mamede, Daniel Ramos, Eli Claggart, Jane Hsieh, Jenny Liang, Jenna DiVincenzo, Jessica Hyunh, Kaia Newman, Kyle Liang, Luke Dramko, Manisha Mukherjee, Nadia Nahar, Nikitha Rao, Parv Kapoor, Rex Chen, Ritam Dutt, Paulo Santos, Sagar Bharad-

waj, Samantha Phillips, Sanjith Athlur, Soham Pardeshi and Vasu Vikram for the countless fun times, game nights and trips that we shared together. I will always remember all the memories we had together, and I am so grateful for everyone's support. I would also like to thank my high school friends for all the fun memories and times together both in person and over Discord: Hazen O'Malley, Sam Sherrill and Stuart McClintock. Finally, I would also like to thank my Facebook friends: Arjun Subramonian, Chantal Shaib, Morris Alper and Harshit Sikchi for all the fun memories and time we spent together in Paris. That summer will always hold a special place in my heart. This list is incomplete; there are so many others who have made a difference in my PhD journey, and I am so grateful for the entire CMU community and the impact it has had on me and my growth.

Last, but most importantly, I would like to thank my family for their unwavering love and support throughout my PhD. I am so grateful for their encouragement, guidance, and love, and I am so thankful for everything they have done for me. I would like to thank my parents, Jyoti and Sandeep Jain, for their endless love and support, and for always believing in me. Finally, I would like to thank my grandfather Surinder Kumar Jain for his unwavering love and support, and for always being there for me. Even though he is no longer with us, I know that he is always watching over me, and I am so grateful for everything he has done for me.

Contents

List of Figures

List of Tables

1 Introduction

Software testing is a critical component of the software development process. A high-quality test suite can be instrumental in finding inconsistencies between a system's specifications and its implementation. These test suites ideally execute all code paths (high code coverage), and catch regressions in the code under test that a developer might introduce (high mutation score) [46, 132]. However, writing high quality tests can be time-consuming [17, 18] and is often either partially or entirely neglected. This has led to extensive work in automated test generation, including both classical [15, 22, 41, 45] and neural-based methods [41, 133, 139]. For this thesis, I focus both automatically generating white box unit tests and improving adequacy metrics for evaluating white unit tests (where the goal is to test individual classes or functions).

Automated test generation approaches, such as EvoSuite [45] and Pynguin [89] can automatically generate high-coverage tests. However, the generated tests are often hard to read and do not look like human written tests [105]. Maintaining and checking these automated test suites requires time and effort from developers, hindering adoption of search based automated test generation techniques [22]. Meanwhile, language models trained on code have made major strides in generating human-like, high-quality functions based on their file-level context and show some promise in generating more readable tests [16, 25, 47, 97]. Test code is natural [56], enabling language models to learn common, routine patterns present in source code.

However, approaches directly applying language models to unit test generation are limited [25, 47, 97]. Language models struggle with hallucination (invoking methods in generated tests that are not in the source file) [81, 83]. They also often fail to invoke internal methods present in the file under test, but not widely available and documented on the internet [83, 95]. To see why, consider how developers write unit tests. To generate a unit test, a developer must understand the code under test, including how to set up the necessary objects, invoke the method under test, and check some property about the code under test [12]. Language models are pretrained on open source code and typically consider source tokens immediately before the generated code. Test code often consists of references to internal API methods, and global and static variables, not seen at pretraining time; without this distant context, language models are not capable of generating correct test cases. Furthermore, current approaches applying language models to test generation do not leverage execution feedback inherent in software testing (compilation, passing, and coverage), which is the reason for the high coverage of classical test generation approaches [45, 68, 116]. This motivates combining existing local context that language models use with distant context in the form of source method code and execution data.

In my work, I target two central tasks: unit test generation and mutation testing. Figure 1.3 shows an example of both unit test generation and mutation testing. Given a partially complete

```
1  public class Bank {
2    public String methodName() {...}
3    ...
4  }
5  <|codetestpair|>
6  public class BankTest {
7    @Test
8    public void FirstTest() {...}
9    ...
10   @Test
11   public void Test_k() {
12     assertNotNull(Bank());
13   }
14   ...
15   @Test
16   public void LastTest() {...}
17   @Test
18   public void ExtraTest() {...}
19 }
```

Figure 1.1: Unit test generation

```
1  public RegularTimePeriod next() {
2    Hour result;
3  -  if (this.hour != LAST_HOUR_IN_DAY) {
4  +  if (this.hour > LAST_HOUR_IN_DAY) {
5      result = new Hour(this.hour + 1,
       this.day);
6    }
7    ...
8  }
9
10 public void testNext() {
11   Hour h = new Hour(1, 12, 23, 2000);
12   h = (Hour) h.next();
13   assertEquals(2000, h.getYear());
14   ...
15 }
```

Figure 1.2: Mutation testing

Figure 1.3: Two software testing tasks. Unit test generation involves generating the first test method, last test method, and extra test method, along with test completion. Predictive mutation testing consists of a source method, mutant (lines 3 and 4) and test method, to predict whether the test kills the mutant. Both tasks require *non-local source context* in addition to test code.

test file and its corresponding code file, the goal of *unit test generation* is to generate the next test method. Developers can use test generation to produce an entire test suite or add tests to an existing test suite to test new functionality. The goal of *mutation testing* is to measure whether a test suite can detect synthetic bugs (mutants). For the synthetic bugs that are not detected by a test suite, the existing test suite can be improved by generating new tests that detect these bugs.

Prior work applying language models to both unit test generation and mutation testing misses the relationship between code and tests. Researchers adapted code generation models to unit test generation [5, 6, 97], resulting in test generation models not considering the relationship between code and tests. As Figure 1.1 shows, it is challenging for a developer to write a unit test without the code under test; the assert for `Test_k` requires a developer to know how to invoke the `Bank` class. Similarly, prior work in applying language models to mutation testing took limited context [74, 146] such as the mutated line and test name. These approaches also miss the relationship between the mutation and the body of the test method: to predict whether the test passes or fails on the mutated code in Figure 1.2, one needs the test method body, specifically the `assert` statements.

Moreover, evaluations of these approaches have largely been confined to small, self-contained programs that do not reflect the real-world challenges faced by developers, leading to overly optimistic performance metrics [25, 96, 128, 135, 139]. In larger, more complex code bases, average coverage remains below 40%, and issues such as hallucination and import errors persist [62]. This poor performance hinders the adoption of automated test generation at scale, creating a disconnect between high benchmark performance and low performance in practice [19**?**].

To address these challenges, I leverage the joint relationship between code and tests to im-

prove the automated testing techniques, evaluating my techniques on large scale, real-world projects. My collaborators and I show that pretraining language models on a dual objective of code and test generation enables them to outperform existing language models with orders of magnitude more parameters and training budgets. We also show that this joint relationship between code and tests can be used to enhance state-of-the-art predictive mutation testing techniques, where we model mutants as a token level diff, and present the model with both the source and test method during fine-tuning. I introduce a novel approach that leverages both execution feedback from running the tests and the coverage report generated to iteratively refine test suites. Unlike prior approaches, this scales to large code bases in a cost-effective way because we collect and iterate on execution feedback at the file level rather than the method level. We show that this approach can significantly improve test generation, with greater than 10% improvement in both line coverage and mutation score. The coupling between code and test files can also be used to create better benchmarks for test generation. By modeling test generation at the file level rather than the method level, we can more accurately capture test generation at scale, with the goal to generate an entire test suite for a file under test rather than generate a test method for a self-contained source method.

1.1 Thesis Statement

My thesis statement is:

We exploit the structure of test code and close relationship between code and test files to improve the evaluation and application of language models to software testing in both pretraining and fine-tuning. This insight can (a) generate useful unit test cases, (b) identify weaknesses in existing test suites, (c) build more realistic test generation benchmarks, and (d) generate test suites for large scale projects.

1.2 Evaluation Metrics and Benchmarks

Improving the evaluation of test generation approaches is a core part of my thesis. I do this in two ways: by leveraging metrics that align with end user experience and by introducing a new benchmark for evaluating test generation approaches that is sourced from large scale open source repositories.

For evaluating unit test generation, our goals include generating tests that both look like developer tests and achieve high code coverage. Tests that look very different from developer tests are harder to maintain [33], and test suites with high coverage are more likely to catch bugs [76]. Prior work on language model unit test and test suite generation [96, 128, 129] evaluated their approaches on lexical metrics such as CodeBLEU [117] and ROUGE [82] score. These metrics quantify how close generated tests look to developer written tests. However, these metrics do not capture all nuances of high quality tests; a test that does not compile or check interesting properties can still have very high CodeBLEU and ROUGE scores. In my work [116], I extend the evaluation of test generation approaches to also include runtime metrics, including what percentage of generations compile, pass the test suite, and add coverage. In my final work (Chapter 6) I

add mutation score, a test adequacy metric more correlated with faults than code coverage [107].

Another goal of evaluating unit test generation is to measure performance on real-world projects. Existing test generation benchmarks [25, 135] are primarily sourced from small programming problems or LeetCode. Unfortunately, these problems do not align with the challenges faced by developers in the real-world; performance on such benchmarks is nearly saturated, with GPT-4o obtaining over 85% coverage on TestEval (an existing test generation benchmark sourced from LeetCode) [135]. As a part of my thesis, I introduce a new test generation benchmark, TESTGENEVAL, sourced from 11 large scale open source projects (3,523-78,287 stars) [62]. TESTGENEVAL consists of long code and test files (average of 1157 LOC and 943 LOC respectively), with high coverage gold test suites to compare against (average coverage of over 80%). As a result, model performance is significantly lower, with even state-of-the-art models achieving less than 40% coverage [62]. This motivated me to work on an agentic approach to unit test generation that scales to these large code bases (Chapter 6).

For the mutation testing task, our goals include measuring the time cost of using our tool when we show the developer no false positives, along with measuring how well our tool performs on non-trivial mutants (a weakness of prior approaches [74]). Multiple studies [66, 93] show developers are far less likely to adopt tools with a high false positive rate, as false positives waste valuable developer time inspecting and fixing non-existent bugs. I add to the evaluation of existing work [74, 146] by considering a setting where the tool checks all predicted undetected mutants to avoid showing the developer false positives. Our checked setting eliminates false positives entirely, allowing us to quantify time saved in a likely setting where our tool would be deployed. We also add additional evaluation that considers the efficacy of our tool on non-trivial mutants (mutants where only a subset of the tests in the test suite detect the mutant). These more challenging cases, are ones we care about more. A tool that can only detect trivially detected mutants has very limited practical utility. We show that in these cases, the performance difference between our tool and existing tools is even more pronounced than the overall performance difference.

1.3 Contributions

I propose a set of techniques that all exploit tests relationship with source code and execution data, not considered in prior work. My projects prove that this source context is helpful in both pretraining and fine-tuning software testing models. I also improve the state of evaluation for both unit test generation and mutation testing, by introducing new metrics that align with end user experience and a new unit test generation benchmark sourced from large, real-world projects. Finally, I develop a test generation agent that scales to the large repositories in my benchmark, and show how execution feedback can be used to improve the quality of generated test suites.

My thesis will make the following contributions:

1. It presents a new method for pretraining models for test generation, that considers the relationship between source code and test code.
2. It provides an approach to automatically classify mutants as detected or undetected without executing the test suite by leveraging additional *test* context.

3. It evaluates all provided techniques with metrics and experiments that are practically meaningful developers, not considered in prior work.
4. It introduces a benchmark for evaluating test generation approaches that is sourced from large scale open source repositories and thus more closely resembles real-world test generation.
5. It demonstrates the effectiveness of adding execution context to test generation models, which enables us to generate high quality test suites for large scale projects.

Parts of this thesis have been published in peer reviewed venues:

- **CAT-LM: Training Language Models on Aligned Code And Tests**, Nikitha Rao*, Kush Jain*, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn, in International Conference on Automated Software Engineering (ASE), 2023 [116].
- **Contextual Predictive Mutation Testing**, Kush Jain, Uri Alon, Alex Groce, and Claire Le Goues, in Foundations of Software Engineering (FSE), 2023 [61].
- **TestGenEval: A real-world Unit Test Generation and Test Completion Benchmark**, Kush Jain, Gabriel Synnaeve, and Baptiste Roziere, in International Conference on Learning Representations (ICLR), 2024 [62].

Additionally, my final chapter is under submission:

- **TestForge: Feedback-Driven, Agentic Test Suite Generation**, Kush Jain*, and Claire Le Goues [60].

1.4 Outline

The rest of the thesis is structured as follows. In Chapter 2 I discuss the related work in test generation, mutation testing, machine learning, test benchmarking and a review of the literature. Chapter 3 describes my work on pretraining language models for test generation by leveraging the tight coupling between code and tests. Chapter 4 describes my work on improving predictive mutation testing by incorporating source and test method context. Chapter 5 describes a large test generation benchmark I built to enable a more realistic evaluation of unit test generation capabilities. Chapter 6 presents an agentic approach to test generation that scales to the large test and code bases I introduced in my test generation benchmark. Finally, Chapter 7 concludes the thesis.

2 Background and Related Work

In this chapter, I discuss important background information and related work for this dissertation. Software testing is a core component of software development, with high quality tests providing some assurance that software behavior matches the desired specification. The goal of a software testing is to validate the behavior of written code against a desired specification. When the specification and code do not match, the tests should ideally fail, indicating a possible bug in the code under test. There are different types of software testing, ranging from system level tests which validate high level behavior of an entire software system to unit tests, which validate the behavior of each component in isolation. For this dissertation, I primarily target unit test generation. Another challenge with software testing measuring the quality of generated tests. I explore how we can make mutation testing (a stronger test adequacy metric) than code coverage more scalable. All of my work involves language models, a statistical model of the future based on past context. Language models have high potential in software testing, helping both automate test generation and improve the efficiency of mutation testing. I discuss the background for each of these topics Section 2.1 and related work for automated testing, mutation testing and test generation evaluation in Section 2.2.

2.1 Background

This section provides an overview of important background information that my thesis is built upon. Specifically, I discuss unit test generation (Section 2.1.1), mutation testing (Section 2.1.2), and language models (Section 2.1.3).

2.1.1 Unit Test Generation

Unit test generation is the process of creating tests for individual units of software (like functions or classes) to verify they work according to a specification [73]. It is common in high-quality development; unit testing helps catch bugs early and helps ensure the correctness of each component in a system. It also has the potential to catch future defects through regression testing.

In development workflows, unit testing is commonly integrated into continuous integration/-continuous deployment (CI/CD) pipelines. In CI/CD pipelines, commits trigger an automated run of the unit test suite, providing feedback to the developer. By catching issues early (ideally before code is merged), teams can fix problems with less effort and avoid defects propagating to later stages.

Developers commonly write unit tests manually [31], crafting input scenarios and expected outcomes for each code unit. Manual test writing, however, is time-consuming and prone to human error [17, 18]. Automated unit test generation attempts to address these challenges by using tools to produce test cases automatically [45, 89]. This saves developers substantial time, allowing them to focus on other aspects of the software engineering lifecycle.

Automated unit test generation offers several benefits over manual testing. Automated test suites can help increase coverage by exploring combinations of inputs that developers might not consider, and helping expose future bugs [113]. It can also better support CI/CD integration, with companies integrating automated unit test generation into their pipelines [11, 14]. These automated testing tools provide substantial value at scale, saving developer time and improving software quality.

2.1.2 Mutation Testing

Mutation testing [39] is the process of synthetically introducing faults into programs and measuring the effectiveness of tests in catching them. A set of program transformations, known as "mutation operators" take regular code and create buggy copies of it. Similar transformations are also used in code migration and refactoring [72, 114]. Mutation operators vary [30, 50, 67], but some common operators include negating conditions (`if (a)` to `if (!a)`), replacing arithmetic operators (`a + b` to `a - b`), replacing relational operators (`a < b` to `a > b`), and flipping conditionals (`a == b` to `a || b`). Each time one of these rules is applied to a program, a new *mutant* is created, each differing only slightly from the original program.

Test adequacy is measured by running the entire test suite on each mutant; the goal is a test suite that detects all mutants, increasing confidence that the suite would detect unintentional bugs as well. Mutation score, or the ratio of detected mutants to total mutants, provides a rough measure of test adequacy, outperforming code coverage in terms of correlation with real-world fault detection [68, 107]. Mutation testing has seen some industry adoption [109**?**]. Prominent recent uses at Facebook and Google apply it only to changed code at commit-time, which still requires large amounts of idle compute [110] because of the massive computational expense of running it over an entire codebase.

2.1.3 Language Models

Language Models (LLMs) [85, 117] are predictive statistical models. They are designed to predict the next token (word or symbol) in a sequence based on prior context. Prior to transformers, sequence modeling leveraged recurrent neural networks (RNNs) [28, 57]. RNNs process sequences token by token, with a vector representation of the state of the sentence at each step. RNN have no inherent limit on context length in theory, but in practice vanishing gradients were a problem (earlier tokens would be "forgotten" by the RNN). Gated RNN architectures like Long Short-Term Memory (LSTM) [57] helped mitigate this issue, by introducing gates to remember and forget relevant information. However, even with these updates RNNs continued to struggle with long sequences, while also not scaling well due to their sequential processing of tokens.

Transformers [131] helped overcome many of these limitations. Transformers use self-attention to handle sequence relationships, where each token in a sequence can attend to every other token

in parallel. The model learns weights for how other tokens relate to each weight, allowing words to take into context long range dependencies. This architecture was refined further, with innovations such as multi-headed attention and positional embeddings, laying the foundation for Large Language Models (LLMs).

LLMs build upon the transformer architecture at a much larger scale. An LLM commonly consists of many transformer blocks (self-attention + feed forward layers) stacked together, with model sizes ranging from hundreds of millions to tens or hundreds of billions of parameters. These models are also trained differently than traditional language models. LLMs are typically pretrained on a large corpus of textual data to predict the next token given the prior context. Through this process, the model gradually learns syntax, semantics, facts, and even some reasoning abilities encoded in the training data [23, 38, 103]. After pretraining, many LLMs undergo fine-tuning or alignment phases: for instance, GPT-4 [102] was trained as a multi-modal model (accepting text and images) and then aligned with human feedback to improve factuality and adherence to desired behaviors.

2.2 Related Work

I also discuss important related work in the areas of automated test generation (Section 2.2.1), reducing the cost of mutation testing (Section 2.2.2) and evaluating test generation approaches (Section 2.2.3). This related work is the foundation for my thesis, and provides context for the contributions I make in this dissertation.

2.2.1 Automated Unit Test Generation

Motivated by the time-consuming nature of manual test writing [17, 18], researchers have developed automated test generation techniques to help developers write tests more efficiently. These techniques can be broadly categorized into classical test generation, neural test generation and most recently test generation agents.

Classical Test Generation

Classical test generation techniques employ both black-box and white-box techniques to generate test inputs and test code. Random/fuzzing techniques such as Randoop [104], aflplusplus [44] and honggfuzz use coverage to guide generation of test prefixes. Property testing tools such as Korat [21], QuickCheck [29] and Hypothesis [91] allow a developer to specify a set of properties and subsequently generate a suite of tests that test the specified properties. PeX [126] and Eclipser [27] use dynamic symbolic execution to reason about multiple program paths and generate interesting inputs. The core issue with fuzzing and classical test generation techniques is their reliance on program crashing or exceptional behavior in driving test generation [41], which limits the level of testing they provide. EvoSuite [45] addresses these challenges by using mutation testing to make the generated test suite compact, without losing coverage. However, EvoSuite generates tests that look “unnatural”, and significantly different from human tests, suffering from both stylistic and readability problems [22, 33, 118]. This motivates using language

models over search based test generation approaches in my thesis, as language models generate much more "natural" and human readable tests than classical approaches.

Neural Test Generation

Neural test generation methods leverage language models to generate more natural and human understandable tests. ConTest[133] makes use of a generic transformer model, using the tree representation of code to generate assert statements. ATLAS [139], ReAssert [142], AthenaTest [128] and TOGA [41] extend this work by leveraging the transformer architecture for this task. They show that their generated asserts are more natural and preferred by developers when comparing against existing tools such as EvoSuite. TeCo [96] expands the scope of test completion by completing statements in a test, one statement at a time. They leverage execution context and execution information to inform their prediction of the next statement, outperforming TOGA and ATLAS on a range of lexical metrics. While these neural approaches solve many of the readability issues of classical test generation approaches, they focus on generating individual statements in a test, which offers significantly less time saving benefits than generating entire tests. The limited context and scope of prior work motivates my work on generating entire test suites rather than components of an individual test.

Test Generation Agents

Recently, there has been extensive work on developing agents for software testing tasks. Agentic approaches are fundamentally different from neural approaches, because they allow LLMs to autonomously interact within an environment, calling tools and acting on feedback from the environment. ChatUniTest [26] and MuTAP [34] target focal method test 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 [111] and HITS [138] extend the prior work by adding dependency analysis and error reports as part of the feedback, but still suffer from cost issues due to their method level approach. The high cost and long runtime of prior approaches motivates my work that generates test suites at the file level and allows the model to iterate on multiple pieces of execution feedback at once.

2.2.2 Reducing the Cost of Mutation Testing

Mutation testing has been shown to improve test suites in ways correlated with real-world fault detection [68, 106]. However, one of its major limitations is its computational cost: test suites must be run on each mutant, in principle. Large-scale systems commonly have hundreds of thousands of mutants [37, 51], since mutants scale with size of the codebase and mutation operators considered. Many approaches have been proposed to tackle the *computational* cost of mutation, including weak-mutation, meta-mutation, mutation-sampling, and predicting which mutants will be killed [71, 99, 130, 146]. Approaches to reducing the cost of mutation analysis were categorized as *do smarter*, *do faster*, and *do fewer* by Offutt and Untch [98]. The *do smarter* approaches

include space-time trade-offs, weak mutation analysis, and parallelization of mutation analysis. The *do faster* approaches include mutant schema generation, code patching, and other methods to make mutants run faster. Finally, the *do fewer* approaches try to reduce the number of mutants examined, and include selective mutation and mutant sampling.

Techniques for predictive mutation testing [74, 92, 146] use machine learning to predict whether a test or a test suite will detect a mutant without actually running those tests (a *do smarter* approach to tackling the computational cost of mutation testing). One limitation of the first ML-based approach for mutation testing prediction [146] is that its performance degrades significantly when it is not trained/evaluated on mutants that are not covered (executed) by any of the tests in the test suite [7]. Uncovered mutants are trivially undetected by a test suite, since a test cannot fail due to a bug on a line it does not execute. They are thus not interesting for the task of predictive mutation testing. Seshat [74] achieves higher accuracy with lower overhead by exclusively using information about the source code and mutation itself (source method, test method, and mutated line). However, even Seshat suffers from low performance, due to missing context in the test setup and test assertions. Motivated by this, we introduce a technique that takes in the entire source method and test method as context for predictive mutation testing.

2.2.3 Evaluating Test Generation

One of the challenges with automated test generation approaches is evaluating techniques test generation capabilities. Recently, there has been more effort to evaluate language models software testing capabilities, however these benchmarks typically still consist of small, self-contained projects. Common code generation benchmarks such as HumanEvalFix [25], MBPP Plus [13] and APPS [55] can easily be adapted to test generation tasks. TestEval [135] measures test generation capabilities for LeetCode problems of varying difficulties. Bhatia et al. [20] measure test generation performance for modular code without external dependencies. While these benchmarks provide important execution metrics, the small size of these problems and solutions does not mirror realistic test generation.

There are also repository level test completion benchmarks, however these benchmarks often lack realistic execution metrics. TeCo [96] and ConTest [133] provides a benchmark for completing unit tests, but only measures lexical metrics such as BLEU [108] and ROUGE scores [82]. ATLAS [139] and TOGA [41] provide large-scale benchmarks but for completing assertions rather than generating entire tests. SWT-Bench [94] provides a benchmark for test method generation targeted as bug fixing PRs. Their task is also adjacent but measuring a specialized part of software testing (generating tests that fail on code prior to PR and pass after PR).

More recently, there have been efforts to create executable code benchmarks. SWEBench [65] measures the ability of language models to generate patches for failing PR tests. R2E [63] provides a collection of 400 repositories that can be executed, however they leverage equivalence test harnesses which look structurally different from human written unit tests. CruxEval [52] provides a large set of executable code snippets, however measures execution reasoning ability, which is a subset of the test generation task.

Motivated by these challenges, I introduce a large scale repository level test generation benchmark sourced from 11 popular open source Python projects. Unlike existing respository level benchmarks, I add execution metrics such as code coverage and mutation score that more closely

align with test adequacy and bug finding capabilities [68, 106].

2.3 Conclusion

This chapter presents background related to software testing and LLMs, along with related work for unit test generation, mutation testing and evaluation of test generation approaches. Testing is a critical component of the development process, and developers generally do not like writing tests. This dissertation examines techniques to help automate both test generation and help speed up test adequacy evaluation. I also examine how we can better improve the state of software testing evaluation, with higher quality metrics and benchmarks that more closely align with real-world testing. I present my contributions in subsequent chapters.

3 Training Language Models on Aligned Code And Tests

In this chapter, my collaborators and I leverage the close relationship between code and test files to improve unit test generation.[1] Generating unit tests is a challenging and time-consuming task, where automation has potential to add significant value. Code context is helpful in generating unit tests; developers often look at the code under test when generating tests [12]. Thus, it is not surprising that existing test generation approaches [5, 41, 96] that either take limited source method context or simply complete the tests given the test prefix struggle with method hallucination and static/global variables. Due to their limited context and autoregressive pretraining signal, it is difficult for existing models to overcome these limitations.

We show that both the test prefix and the *entire* source file are important in generating tests. We propose the Aligned Code And Tests Language Model (CAT-LM), a GPT-style language model with 2.7 Billion parameters, trained on a corpus of Python and Java projects. We utilize a novel pretraining signal that explicitly considers the mapping between code and test files when available. We also drastically increase the maximum sequence length of inputs to 8,192 tokens, 4x more than typical code generation models, to ensure that the code context is available to the model when generating test code.

We evaluate CAT-LM against several baselines across two realistic applications: test method generation and test method completion. For test method generation, we compare CAT-LM to both human written tests as well as the tests generated by StarCoder [80] and, the CodeGen [97] model family, which includes mono-lingual models trained on a much larger budget than ours. We also compare against TeCo [96], a recent test-specific model, for test completion. CAT-LM generates more valid tests on average than StarCoder and all CodeGen models, and substantially outperforms TeCo at test completion. Our evaluation is more comprehensive than prior work, adding runtime metrics such as compilation, passing, and coverage to the standard lexical metrics of CodeBLEU and ROUGE. These metrics more closely align with the practical utility of generated tests, as developers expect generated tests to compile, pass and cover new code.

Our results highlight the merit of combining the power of large neural methods with a pretraining signal based on a core insight in my thesis, the importance of the relation between code and test files. In summary the contributions for this chapter are:

- A corpus of 1.1M code-test file pairs along with 14.4M Java and Python files across 196K open-source projects. We believe this corpus will be useful for many testing-related tasks.

[1] Work that appeared in ASE 2023 [116]

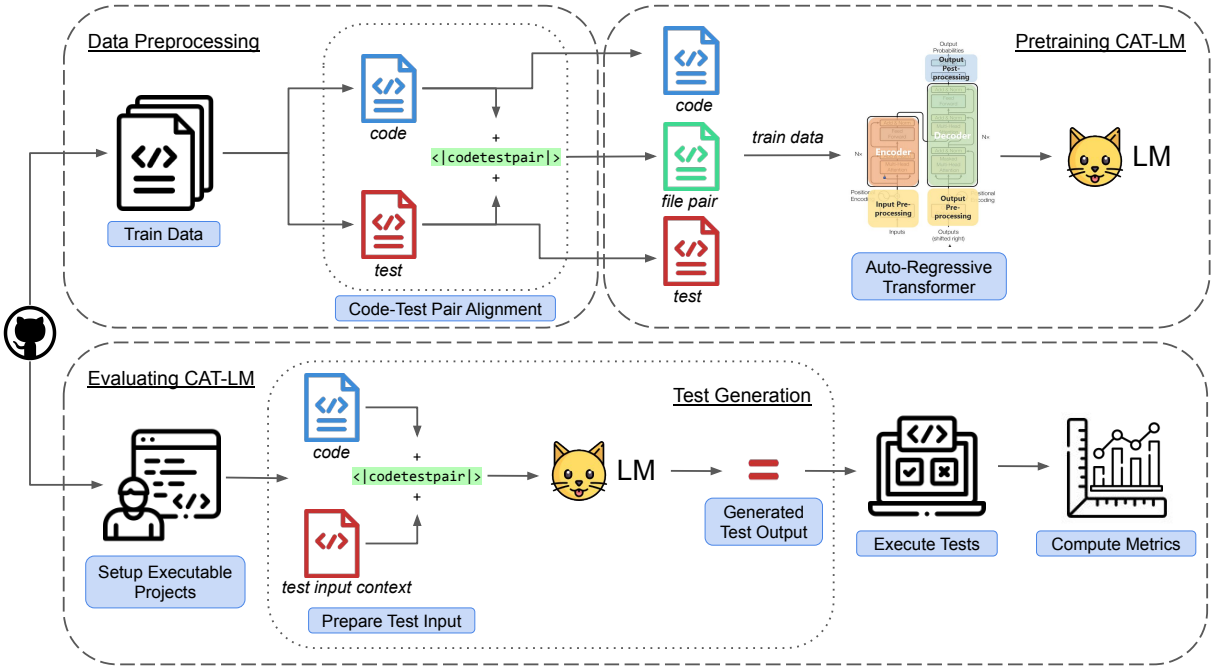


Figure 3.1: Approach overview. We extract Java and Python projects with tests from GitHub and heuristically align code and test files (top), which, along with unaligned files, train CAT-LM, a large, autoregressive language model. We evaluate CAT-LM's generated tests on a suite of executable projects (bottom), measuring its ability to generate syntactically valid tests that yield coverage comparable to those written by developers.

- CAT-LM, the first pretrained LLM that models aligned code and test files from Java and Python projects on GitHub.
- The testing framework used to evaluate the tests generated by CAT-LM.
- An evaluation of CAT-LM with baselines on downstream tasks such as test method generation and test completion.

Since the time of publication, this work has been superseded by TestForge (Chapter 6), but the conceptual contribution of the work remains relevant. The model CAT-LM itself is now outdated, as newer models such as Llama 3.1 [**?**] and Gemini [125] have been released, which are much larger and more powerful than CAT-LM. However, both the aligned dataset of code and test filepairs and pretraining objective still remain relevant and do scale to current larger models. This dataset has helped improve test generation capabilities of modern models, and is incorporated into the training of multiple state-of-the-art models.

3.1 Overview

CAT-LM is a GPT-style model that can generate tests given code context. Figure 3.1 shows an overview of our entire system, which includes data collection and preprocessing (detailed in Section 3.3.1), pretraining CAT-LM (Section 3.4), and evaluation (Section 3.5).

We first collect a corpus of approximately 200K Python and Java GitHub repositories, focusing on those with at least 10 stars. We split these at the project level into a train and test set (Section 3.3.1). We filter our training set following CodeParrot [141] standards (including deduplication), resulting in $\sim$15M code and test files. We align code and test files using a fuzzy string match heuristic (Section 3.3.2).

We then prepare the training data, comprising of the code-test file pairs, paired with a unique token (`<|codetestpair|>`), as well as unpaired code and test files. We tokenize the files using a custom-trained sentencepiece tokenizer [3]. We then determine the appropriate model size, 2.7B parameters based on our training budget and the Chinchilla scaling laws [58]. We use the GPT-NeoX toolkit [2] enhanced with Flash Attention [35]

To pretrain CAT-LM using an auto-regressive (standard left-to-right) pretraining objective that captures the mapping between code and test files, while learning general code and test structure.

Finally, we evaluate CAT-LM on the held-out test data. We manually set up all projects with executable test suites from the test set to form our testing framework. We prepare our test inputs for CAT-LM by concatenating the code context to the respective test context for test generation. The test context varies based on the task. We assess our model's ability to generate (1) the first test method, (2) the last test method, add (3) an additional, new test to an already complete test suite. We also evaluate completing a statement within a test function. We tokenize prepared input and task CAT-LM with sampling multiple (typically 10) test outputs, each consisting of a single method. We then attempt to execute the generated tests with our testing framework and compute metrics like number of generated tests that compile and pass, along with the coverage they provide, to evaluate test quality.

Test generation with code context

```
public class Bank {
    public String methodName() {...}
    ...
}
<|codetestpair|>
public class BankTest {
    @Test
    public void FirstTest() {...}
    ...
    @Test
    public void Test_k() {
        assertNotNull(Bank());
    }
    ...
    @Test
    public void LastTest() {...}
    @Test
    public void ExtraTest() {...}
}
```

Figure 3.2: Evaluation tasks, with code context shown for completeness: test generation for the first test method, last test method, and extra test method, along with test completion for Java.

3.2 Tasks

We describe two tasks for which CAT-LM can be used, namely test method generation (with three settings) and test completion. Figure 3.2 demonstrates the setup for all tasks including code context.

3.2.1 Test Method Generation

Given a partially complete test file and its corresponding code file, the goal of *test method generation* is to generate the next test method. Developers can use test generation to produce an entire test suite, or add tests to an existing test suite to test new functionality. We evaluate three different settings, corresponding to different phases in the testing process, namely generating (1) the *first test* in the file, representing the beginning of a developer's testing efforts. In this setting, we assume that basic imports and high-level scaffolding are in place, but no test cases have been written, (2) the *final test* in a file, assessing a model's ability to infer what is missing from a near-complete test suite. We evaluate this ability only on test files that have two or more (human-written) tests to avoid cases where only a single test is appropriate, and (3) an *extra* or additional test, which investigates whether a model can generate new tests for a largely complete test suite. Note that this may often be unnecessary in practice.

Table 3.1: Summary statistics of the overall dataset.

Category	Attribute	Python	Java	Total
Project	Total	148,605	49,125	197,730
	Deduplicated	147,970	48,882	196,852
	W/o Tests	84,186	15,128	99,314
	W/o File pairs	108,042	23,933	131,975
Size (GB)	Raw	123	157	280
	Deduplicated	53	94	147
Files	Total	8,101,457	14,894,317	22,995,774
	Filtered	7,375,317	14,698,938	22,074,255
	Deduplicated	5,101,457	10,418,609	15,520,066
	Code	4,128,813	8,380,496	12,509,309
	Test	972,644	2,038,113	3,010,757
	File pairs	412,881	743,882	1,156,763
	Training	4,688,576	9,674,727	14,363,303

3.2.2 Test Completion

The goal of *test completion* is to generate the next statement in a given incomplete test method. Test completion aims to help developers write tests more quickly. Although test completion shares similarities with general code completion, it differs in two ways: (1) the method under test offers more context about what is being tested, and (2) source code and test code often have distinct programming styles, with test code typically comprising setup, invocation of the method under test, and assertions about the output (the test oracle).

3.3 Dataset

This section describes dataset preparation for both training and evaluating CAT-LM. Table 3.1 provides high-level statistics pertaining to data collection and filtering. In Section 3.4, we describe the training process, including model architecture, data preparation and model output.

3.3.1 Data Collection

We use the GitHub API [1] to mine Python and Java repositories that have at least 10 stars and have new commits after January 1st, 2020. Following [10] and [86], we also remove forks, to prevent data duplication. This results in a total of 148,605 Python and 49,125 Java repositories with a total of $\sim$23M files (about 280 GB). We randomly split this into a train and test set, ensuring that the test set includes 500 repositories for Python and Java each.

3.3.2 Training Data Preparation

We first remove all non-source code files (e.g., configuration and README files) to ensure that the model is trained on source code only. We then apply a series of filters in accordance with CodeParrot's standards [141] to minimize noise from our training signal. This includes removing files that are larger than 1MB, as well as files with any lines longer than 1000 characters; an average line length of $>$100 characters; more than 25% non-alphanumeric characters, and indicators of being automatically generated. This removes 9% of both Python and Java files. We deduplicate the files by checking each file's md5 hash against all other files in our corpus. This removes approximately 30% of both Python and Java files.

We extract code-test file pairs from this data using a combination of exact and fuzzy match heuristics. Given a code file with the name `<CFN>`, we first search for test files that have the pattern `test_<CFN>`, `<CFN>_test`, `<CFN>Test` or `Test<CFN>`. If no matches are found, we perform a fuzzy string match [4] between code and test file names, and group them as a pair if they achieve a similarity score greater than 0.85. If multiple matches are found, we keep the pair with the highest score.

Following file pair extraction, we prepare our training data by replacing the code and test files with a new file that concatenates the contents of the code file and the test file, separating them with a unique `<|codetestpair|>` token. This ensures that the model learns the mapping between code and test files from the pretraining signal. Note that we always combine these files starting with the code, so the model (which operates left-to-right) only benefits from this pairing information when generating the test. We additionally include all the other code and test files for which we did not find pairs in our training data, which results in 4.7M Python files and 9.7 Java files. We include these unmatched files to maximize the amount of data the model can learn from. Figure 3.3 summarizes the distribution of files in the training data along with sample code snippets for each type of file.

Distribution of files and file pairs: Figure 3.4 summarizes the distribution of files in projects with respect to their star count. We observe a decreasing trend in not just the number of code files and test files, but also the file pairs. Upon manual inspection of a few randomly selected projects, we find that popular projects with a high star count tend to be better-tested, in line with prior literature [75, 122]. Note that we normalize the plot to help illustrate trends by aggregating projects in buckets based on percentiles, after sorting them based on stars. The data distribution varies between Python and Java: Python has approximately 3x more projects than Java, but Java has roughly twice as many code-test file pairs.

3.3.3 Test Data Preparation and Execution Setup

To prepare our test data, we first excluded all projects without code-test file pairs. This resulted in a total of 97 Java and 152 Python projects. We then attempted to set up all projects for automated test execution.

Execution Setup for Java: Projects may use different Java versions (which include Java 8, 11, 14, and 17) and build systems (mostly Maven and Gradle). We manually set up Docker images for each combination. We then attempted to execute the build commands for each project in a container from each image. We successfully built 54 out of the 97 Java projects, containing 61

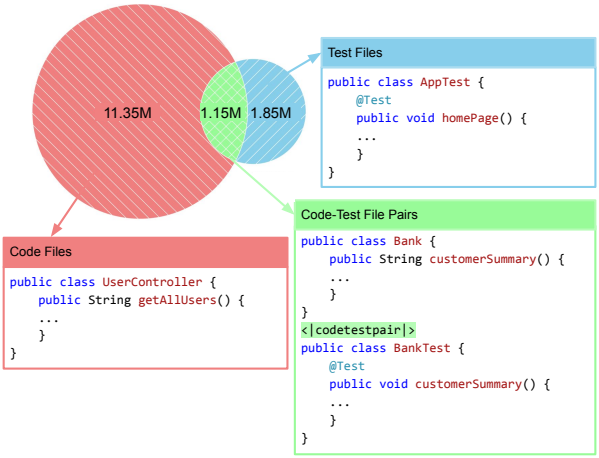


Figure 3.3: Distribution of files with sample code snippets

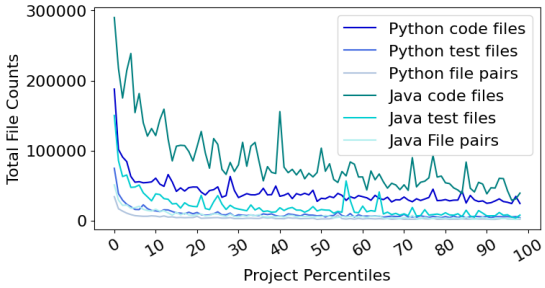


Figure 3.4: Distribution of files in projects sorted by GitHub stars, normalized by percentiles

code-test file pairs.

Execution Setup for Python: We manually set up Docker containers for Python 3.8 and 3.10 with the `pytest` framework and attempted to run the build commands for each project until the build was successful. We successfully built 41 of the 152 Python projects, containing 1080 code-test file-pairs.

We further discarded all *pairs* within these projects with only a single code method or a single test method to ensure that code-test file-pairs in our test set correspond to nontrivial test suites. We additionally require the Java and Python projects to be compatible with the `Jacoco` and `coverage` libraries respectively. This leaves a total of 27 code-test file pairs across 26 unique Java projects and 517 code-test file pairs across 26 unique Python projects. In Python, we randomly sampled up to 10 file pairs per project to reduce the bias towards large projects (the top two projects account for 346 tests) leading to a final set of 123 file pairs across 26 unique Python projects. Note that we reuse these Docker containers in our testing framework (See Section 3.5.1).

3.4 CAT-LM

This section describes the details for preparing the input, pretraining CAT-LM and generating the outputs.

3.4.1 Input Representation for Pretraining CAT-LM

We use the corpus of 14M Java and Python files that we prepared for the pretraining of our model (see Section 3.3.1). We first train a subword tokenizer [77] using the SentencePiece [3] toolkit with a vocabulary size of 64K tokens. The tokenizer is trained over 3 GB of data using ten random lines sampled from each file. We then tokenize our input files into a binary format used to efficiently stream data during training.

Analyzing the distribution of tokens: Language models are typically constrained in the amount of text they fit in their context window. Most code generation models at the time used a context window of up to 2,048 tokens [97, 143].[2] Our analysis on the distribution of tokens, visualized in Figure 3.5, showed that this only covers 35% of the total number of file pairs. As such, while it may be appropriate for a (slight) majority of individual files, it would not allow our model to leverage the code file's context while predicting text in the test file. This is a significant limitation since we want to train the model to use the context from the code file when generating tests.

Further analysis showed that approximately 82% of all file pairs for Java and Python have fewer than 8,192 tokens. Since the cost of the attention operation increases quadratically with the context length, we choose this cutoff to balance training cost and benefit. Therefore, we chose to train a model with a longer context window of 8192 tokens to accommodate an additional $\sim$550K file pairs. Note that this does not lead to any samples being discarded; pairs with more tokens will simply be (randomly) chunked by the training toolkit.

[2]The average length of a token depends on the vocabulary and dataset, but can typically be assumed to be around 3 characters.

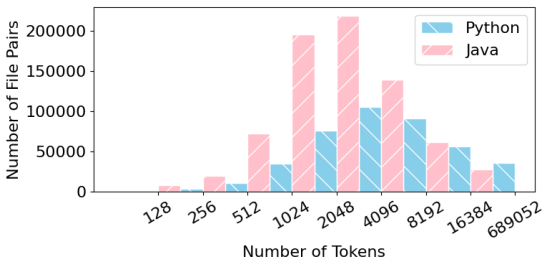


Figure 3.5: Distribution of file pair tokens

3.4.2 Model and Training Details

We determined the model size based on our cloud compute budget of $20,000 and the amount of available training data, based on the Chinchilla scaling laws [58], which suggest that the training loss for a fixed compute budget can be minimized (lower is better) by training a model with ca. (and no fewer than) 20 times as many tokens as it has parameters. Based on preliminary runs, we determined the appropriate model size to be 2.7 (non-embedding) parameters, a common size for medium to large language models [97, 143], which we therefore aimed to train with at least 54B tokens. This model architecture consists of a 2,560-dimensional, 32 layer Transformer model with a context window of 8,192 tokens. We trained the model with a batch size of 256 sequences, which corresponds to ∼2M tokens. We use the GPT-NeoX toolkit [2] to train the model efficiently with 8 Nvidia A100 80GB GPUs on a single machine on the Google Cloud Platform. We trained the model for 28.5K steps, for a total of nearly 60B tokens, across 18 days, thus averaging roughly 1,583 steps per day. We note that this training duration is much shorter than many popular models [97, 127];[3] the model could thus be improved substantially with further training. The final model is named CAT-LM as it is trained on aligned **C**ode **A**nd **T**ests.

3.4.3 Prompting CAT-LM to generate outputs

Since CAT-LM has been trained using a left-to-right autoregressive pretraining signal, it can be prompted to generate some code based on the preceding context. In our case, we task it to either generate an entire test method given the preceding test (and usually, code) file context, or generating a line to complete the test method (given the same). We prompt CAT-LM with the inputs for each task, both with and without code context, and sample 10 outputs from CAT-LM with a "temperature" of 0.2, which encourages generating different, but highly plausible (to the model) outputs. Sampling multiple outputs is relatively inexpensive given the size of a method compared to the context size, and allows the model to efficiently generate multiple methods from an encoded context. We can then filter out tests that do not compile, lack asserts, or fail (since we are generating behavioral tests), by executing them in the test framework. We prepare the outputs for execution by adding the generated test method to its respective position in the baseline test files, without making any changes to the other tests in the file.

[3]The "Chinchilla" optimum does not focus on maximizing the performance for a given model size, only for a total compute budget.

3.5 Experimental Setup

We evaluate CAT-LM's ability to generate valid tests that achieve coverage, comparing against state-of-the-art baselines for both code generation and test completion. We extend prior evaluations of neural test generation approaches [**? ? ?**] by adding runtime metrics to the standard lexical evaluation of tools. We choose to measure runtime metrics because developers are likely to only use an automated test generation approach if the approach generates both compiling and passing tests.

3.5.1 Test Method Generation

The test method generation task involves three different cases: generating the first test, the final test, and an extra test in a test suite (see Section 3.2). We evaluate CAT-LM on test method generation both with code context and, as an ablation, without code context.

Baseline Models

CodeGen is a family of Transformer-based LLMs trained autoregressively (left-to-right) [97]. Pretrained CodeGen models are available in a wide range of sizes, including 350M, 2.7B, 6.1B and 16.1B parameters. These models were trained on three different datasets, starting with a large, predominantly English corpus, followed by a multilingual programming language corpus (incl. Java and Python), and concluding with fine-tuning on Python data only. The largest model trained this way is competitive with Codex [25] on a Python benchmark [97].

For our evaluation, we compare with CodeGen-2.7B-multi, which is comparable in size to our model and trained on multiple programming languages, like our own. We also consider CodeGen-16B-multi (with 16B parameters, ca. 6 times larger than CAT-LM) which is the largest available model trained on multiple programming languages. For all Python tasks, we also compare against CodeGen-2.7B-mono and CodeGen-16B-mono, variants of the aforementioned models fine-tuned on only Python code for an additional 150k training steps.

We also compare the performance of CAT-LM with StarCoder [80], which is a 15.5B parameter model trained on over 80 programming languages, including Java and Python, from The Stack (v1.2). StarCoder has a context window of $8,192$ tokens. It was trained using the Fill-in-the-Middle objective [16] on 1 trillion tokens of code, using the sample approach of randomizing the document order as CodeGen.

Lexical Metrics

Although our goal is not to exactly replicate the human-written tests, we provide measures of the *lexical* similarity between the generated tests and their real-world counterparts as indicators of their realism. Generated tests that frequently overlap in their phrasing with ground-truth tests are likely to be similar in structure and thus relatively easy to read for developers. Specifically, we report both the rate of exact matches and several measures of approximate similarity, including ROUGE [82] (longest overlapping subsequence of tokens) and CodeBLEU [117] score (n-gram overlap that takes into account code AST and dataflow graph). We only report lexical metrics

for our first test and last test settings, as there is no ground truth to compare against in our extra test setting. These metrics have been used extensively in prior work on code generation and test completion [59, 78, 96, 137].

Runtime Metrics

We also report runtime metrics that better gauge test utility than the lexical metrics. This includes the number of generated tests that compile, and generated tests that pass the test suite. We also measure coverage of the generated tests. For first and last tests, we compare this with the coverage realized by the corresponding human-written tests. We hope that this work will encourage more widespread adoption of runtime metrics (which are an important part of test utility), as prior work primarily focuses on lexical similarity [41, 96, 139]. For additional detailed descriptions of all lexical and run-time metrics, results are available in published work [116].

Preparing Input Context and Baseline Test Files

We use an AST parser on the ground-truth test files to prepare partial tests with which to prompt CAT-LM. For first test generation, we remove all test cases (but not the imports, nor any other setup code that precedes the first test); for last test generation, we leave all but the final test method, and for final test generation we only remove code after the last test. We then concatenate the code context to the test context using our delimiter token for the 'with code context' condition.

We additionally obtain coverage with the original, human-written test files under the same conditions, keeping only the first or all tests as baselines for first and last test prediction respectively. Note that there is no baseline for the extra test generation task. For the coverage distribution of human-written tests see published work [116].

Testing Framework

We evaluate the quality of the generated tests using the containers that we set up to execute projects in Section 3.3.3. We insert the generated test into the original test file, execute the respective project's setup commands and check for errors, recording the number of generated tests that compile and pass the test suite (see Section 3.5.1). If the generated test compiles successfully (or, for Python, is free of import or syntax errors), we run the test suite and record whether the generated test passed or failed. We compute code coverage for all passing tests, contrasting this with the coverage achieved by the human-written test cases (when available) as baselines.

3.5.2 Test Completion

Recall the test completion task involves generating a single line in a given test method, given the test's previous lines. We perform our evaluation for test completion under two conditions, with code context and without code context.

Table 3.2: Baseline coverage for human written tests over the given number of file pairs.

Programming Language	Case	Coverage Improvement %	# File Pairs
Python	First test	59.3%	112
	Last test	5.0%	93
	Extra test	0.0%	123
Java	First test	50.5%	27
	Last test	5.3%	18
	Extra test	0.0%	27

Baseline Model

We compare against TeCo [96], a state-of-the-art-baseline on test statement completion that has outperformed many existing models, including CodeT5 [137], CodeGPT [87] and TOGA [41]. TeCo [96] is an encoder-decoder transformer model based on the CodeT5 architecture [137]. TeCo takes the test method signature, prior statements in the test, the method under test, the variable types, absent types and method setup and teardown as input.

Initially, we intended to compare CAT-LM against TeCo on our test set. However, TeCo performs extensive filtering including requiring JUnit, Maven, well-named tests, a one-to-one mapping between test and method under test, and no if statements or non-sequential control flow in the test method. We thus compared CAT-LM against TeCo for 1000 randomly sampled statements from their test set.

Metrics

We compare CAT-LM against TeCo across all lexical metrics (outlined in Section 3.5.1).

3.6 Limitations and Threats

Limitations: One limitation of CAT-LM is our use of flash attention [35]. Flash attention allows us to leverage the NVIDIA A100 architecture to train CAT-LM with a much larger context window (8192 tokens) in the same compute budget. Due to this optimization, fine-tuning CAT-LM on older GPUs is likely to be slow and not advisable.

Threats to Validity: The main internal threat to validity is our implementation of CAT-LM. We used widely available and popular libraries for managing data and building the model to help mitigate this threat. We release our models and implementation for inspection and extension by others. The external threats to validity lie in our dataset of tests and file pairs. We filter out projects that have not been committed to recently and ones with fewer than 10 stars to ensure that we train on up-to-date, well tested code. We also perform standard practices of removing duplicate data to ensure no leakage between our own training and test sets. Since this dataset is sourced from a large number of open-source projects, the results are more likely to generalize.

Another potential threat to external validity is data leakage when compared to existing baselines. It is important to consider that both GPT-4 and CodeGen baselines have likely seen our test set during their pretraining. Similarly, we have likely seen TeCo's test set during our pretraining phase. We tried to avoid data leakage and run TeCo on our test set, however, their extensive filtering process makes this task nearly impossible. This data leakage can inadvertently result in overly optimistic evaluation results, as models are indirectly trained on the same data they are being tested on.

Threats to construct validity lie primarily in our evaluation metrics. We report widely used metrics, i.e., CodeBLEU, ROUGE, compiling generations and passing generations. While these metrics approximate similarity to developer tests, they do not capture all aspects of test quality, for example, whether a test compiles, passes for the code under test and whether it executes all paths in the file under test.

3.7 Results

We report CAT-LM's performance across runtime and lexical metrics for both test method generation and test completion. Additional results can be found in published work [116].

3.7.1 Test Method Generation

Pass Rate

Figure 3.6 shows the number of passing tests generated by each model for Python and Java. Note that these are absolute numbers, out of a different total for each setting.[4]

CAT-LM outperforms StarCoder and all CodeGen models, including ones that are much larger and language-specific in most settings. For Python, all models perform worst in the first test setting, where they have the least context to build on. Nonetheless, equipped with the context of the corresponding code file, our model generates substantially more passing tests than StarCoder (with 15.5B parameters) and the multilingual CodeGen baselines (trained with far more tokens) in both first and extra test setting. Only in the last-test settings do some of the models compete with ours, though we note that their performance may be inflated as the models may have seen the files in our test set during training (the test set explicitly omits files seen by CAT-LM during training). For Java, we find that CAT-LM generates more passing tests than StarCoder and the two multilingual CodeGen models (no Java-only model exists). The difference is most pronounced in the extra test setting, where CAT-LM generates nearly twice as many passing tests compared to StarCoder and the CodeGen baseline models. Overall, despite being undertrained, CAT-LM generates more number of passing tests on average across all settings. Both StarCoder and the CodeGen models do not show significant gains with more parameters or longer contexts (StarCoder can use $8,192$ tokens), highlighting that training with code context is important.

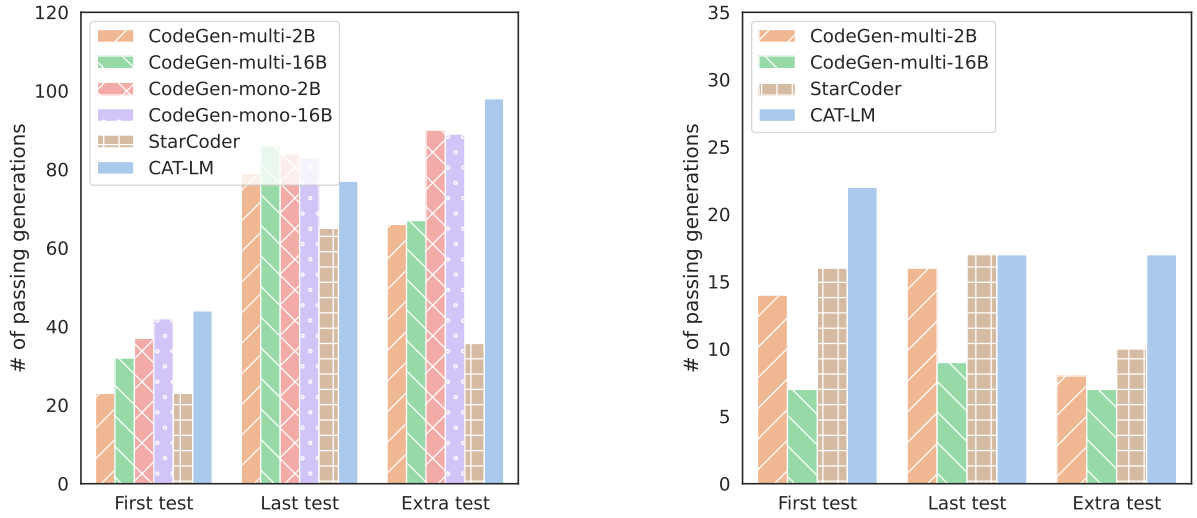
[4]The denominator for each group is the number of file pairs shown in Table 3.2 multiplied by 10, the number of samples per context.

Table 3.3: Lexical and runtime metrics performance comparison for Java on the held-out test set.

	Lexical Metrics			Runtime Metrics	
Model	**CodeBLEU**	**XMatch**	**Rouge**	**Compile**	**Pass**
First Test (Total: Java = 270)					
CAT-LM w Context	41.4%	15.4%	60.9%	50	22
CAT-LM w/o Context	37.5%	15.4%	56.5%	9	9
Codegen-2B	35.5%	7.7%	56.8%	24	14
Codegen-16B	42.2%	7.7%	61.8%	25	7
StarCoder	44.6%	10.9%	62.2%	28	16
Last Test (Total: Java = 180)					
CAT-LM w Context	55.4%	20.8%	70.8%	54	17
CAT-LM w/o Context	53.6%	20.8%	68.9%	33	14
Codegen-2B	51.7%	13.0%	69.2%	43	16
Codegen-16B	56.5%	14.3%	70.9%	24	9
StarCoder	56.9%	21.0%	69.9%	34	17
Extra Test (Total: Java = 270)					
CAT-LM w Context	–	–	–	41	17
CAT-LM w/o Context	–	–	–	29	20
Codegen-2B	–	–	–	17	8
Codegen-16B	–	–	–	15	7
StarCoder	–	–	–	17	10

Table 3.4: Lexical and runtime metrics performance comparison for Python on the held-out test set.

	Lexical Metrics			Runtime Metrics	
Model	**CodeBLEU**	**XMatch**	**Rouge**	**Compile**	**Pass**
First Test (Total: Python = 1120)					
CAT-LM w Context	21.0%	0.3%	39.4%	384	44
CAT-LM w/o Context	17.7%	0.4%	30.2%	236	31
Codegen-2B	18.2%	0.0%	30.9%	259	37
Codegen-16B	20.8%	0.3%	35.1%	361	42
StarCoder	24.0%	1.8%	38.8%	269	23
Last Test (Total: Python = 930)					
CAT-LM w Context	38.3%	4.8%	54.9%	335	77
CAT-LM w/o Context	33.2%	1.4%	51.9%	350	79
Codegen-2B	36.3%	2.2%	53.2%	326	84
Codegen-16B	37.9%	3.4%	54.0%	349	83
StarCoder	37.6%	4.2%	54.5%	227	65
Extra Test (Total: Python = 1230)					
CAT-LM w Context	–	–	–	380	98
CAT-LM w/o Context	–	–	–	425	104
Codegen-2B	–	–	–	376	90
Codegen-16B	–	–	–	384	89
StarCoder	–	–	–	269	36



(a) Python.

(b) Java.

Figure 3.6: Passing tests by model for Python and Java.

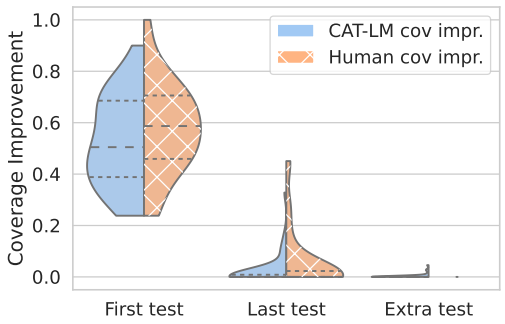
Coverage

Figure 3.7 shows the coverage distribution of CAT-LM, contrasted with that of the human-written tests. For both the first test and last test settings, our model performs mostly comparably to humans, with both distributions having approximately the same median and quartile ranges. The extra test task is clearly especially hard: while our model was able to generate many tests in this setting (Figure 3.6), these rarely translate into *additional* coverage, beyond what is provided by the rest of the test suite, in part because most of the developer-written test suites in our dataset already have high code coverage (average coverage of 78.6% for Java and 81.6% for Python), and may have no need for additional tests. Table 3.2 shows the average human coverage improvement for the first and last test added to a test suite. Note that the average is significantly lower for last test, as baseline coverage is already high for this mode (74.7% for Java and 76.1% for Python).

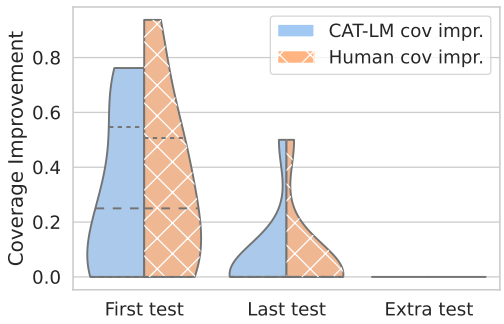
We note that we could not compute coverage for all the file pairs in each setting. We excluded file pairs with only one test from our last test setting to differentiate it from our first test setting. For the first test setting, some baseline files were missing helper methods between the first test and last test in the file, preventing us from computing coverage.

Lexical Similarity

Table 3.4 and Table 3.3 show the lexical similarity metrics results relative to the human-written tests for CAT-LM, both with and without context, along with StarCoder and CodeGen baselines. CAT-LM reports high lexical similarity scores when leveraging code context, typically at or above the level of the other best model, StarCoder (with 15B parameters). This effect is consistent across first and last test generation.



(a) Coverage improvement of our model vs humans for Python.



(b) Coverage improvement of our model vs humans for Java.

Figure 3.7: Coverage improvement of our model vs humans for different languages.

Impact of Code Context

As is expected, CAT-LM heavily benefits from the presence of code context. When it is queried without this context, its performance on lexical metrics tends to drop to below the level of CodeGen-2B, which matches it in size but was trained with more tokens. The differences in lexical metric performance are sometimes quite pronounced, with up to a 9.2% increase in Rouge score and up to a 5.1% increase in CodeBLEU score.

In terms of runtime metrics, code context mainly helps on the first and last test prediction task, with especially large gains on the former. Context does not seem to help generate more passing tests in the extra test setting. This may be in part because the test suite is already comprehensive, so the model can infer most of the information it needs about the code under test from the tests. It may also be due to the test suites often being (nearly) complete in this setting, so that generating additional tests that pass (but yield no meaningful coverage) is relatively straightforward (e.g., by copying an existing test). Overall, these results support our core hypothesis that models of code should consider the relationship between code and test files to generate meaningful tests.

Other Runtime Metrics

Table 3.4 and Table 3.3 also show a comparison between CAT-LM and StarCoder and CodeGen baselines for all runtime metrics. CAT-LM outperforms both StarCoder and the CodeGen baselines in both Python in Java across compiling and passing generations, with CAT-LM typically generating the most samples that compile and pass. The one setting where the CodeGen baselines perform slightly better is in generating more last tests that pass for Python. However, the compile rate of these CodeGen generated tests is significantly lower than those generated by CAT-LM. We note that CodeGen's performance may be inflated in the last test setting, as it may have seen the files from the test set during training.

Table 3.5: Comparison of CAT-LM and TeCo on 1000 randomly sampled statements in their test set.

Model	CodeBLEU	XMatch	Rouge
CAT-LM w/ Context	67.1%	50.4%	82.8%
CAT-LM w/o Context	65.9%	48.9%	82.2%
TeCo	26.7%	13.8%	60.2%

3.7.2 Test Completion

For test completion (see Section 3.2.2 for task definition), we compare CAT-LM against TeCo [96] on the lexical metrics outlined in Section 3.5.1. Specifically, we sample 1000 statements at random from across the test set released by the authors of TeCo, on which we obtain similar performance with TeCo to those reported in the original paper. Table 3.5 shows the results. CAT-LM outperforms TeCo across all lexical metrics, with a 36.6% increase in exact match, 22.6% increase in ROUGE and 40.4% increase in CodeBLEU score. Even prompting CAT-LM with just the test context (i.e., without the code context) yields substantially better results than TeCo. This underscores that providing the entire test file prior to the statement being completed as context, rather than just the setup methods, is helpful for models to reason about what is being tested.

In contrast to the test generation task, code context only slightly helps CAT-LM in this setting, with an increase in CodeBLEU score of 1.2% and increase in exact match accuracy of 1.5%. Apparently, many individual statements in test cases can be completed relatively easily based on patterns found in the test file, without considering the code under tests. This suggests that statement completion is significantly less context-intensive than whole-test case generation. We therefore argue that entire test generation is a more appropriate task for assessing models trained for test generation.

3.8 Conclusion

This chapter illustrates the key insight behind my thesis: the importance of domain specific properties, namely the relationship between code and test files when applying language models to software testing. We introduce CAT-LM, a GPT-style language model with 2.7 Billion parameters that was pretrained using a novel signal that explicitly considers the mapping between code and test files when available. We elect to use a larger context window of 8,192 tokens, 4x more than typical code generation models, to ensure that code context is available when generating tests. We evaluate CAT-LM on both test method generation and test completion, with CAT-LM outperforming CodeGen, StarCoder, and TeCo state-of-the-art baselines, even with CodeGen and StarCoder baselines significantly larger training budgets and model sizes. We show that adding the additional context helps CAT-LM, with code context significantly improving both lexical and runtime metric performance. Overall, we highlight how incorporating domain knowledge, namely the relationship between code and test files, can be used to create more powerful models for automated test generation. While this work is superseded by TestForge (Chapter 6), the dataset and pretraining objective still remain valuable to the community (multiple current state-

of-the-art models incorporate the CAT-LM objective and dataset as part of their training process). This coupling between code and test files can also help improve other software testing tasks such as mutation testing (Chapter 4) and test suite generation (Chapter 6).

4 Contextual Predictive Mutation Testing

In this chapter, my collaborators and I apply the key insight that test code and source code are tightly coupled to automatically detect inadequacies in existing test suites *without* executing the tests.[1] The goal of mutation testing is to improve test quality by finding synthetic bugs (mutants) that existing tests fail to detect. The main limitation of mutation testing is that it is costly to scale (for each synthetic bug introduced, the entire test suite needs to be run). We can significantly reduce test execution time by using language models to automatically predict whether mutants will be detected or not by the test suite (a technique known as predictive mutation testing) [74, 146]. This is because language model inference time is much faster than running the test suite.

While this technique is promising, prior work in predictive mutation testing [74, 146], took limited context such as the test method name and mutated line and thus failed to achieve performance needed for practical use. We leverage the tight coupling between mutated source method code and test code to improve predictive mutation testing techniques. For mutation testing, test bodies have important information such assertions and calls to the method under test.

We introduce MutationBERT, an approach for predictive mutation testing that simultaneously encodes the source method mutation and test method, capturing key *context* in the input representation. MutationBERT learns the relationship between them to predict whether the test will fail on that modified method. To this end, we introduce a novel input representation that encodes each mutation as a token level diff applied to a source method, followed by the corresponding test. We then use a pretrained transformer [131] architecture to encode source and test methods, and further fine-tune it for our task.

We evaluate MutationBERT in both same project and cross project settings, measuring both accuracy and execution time (high accuracy ensures a low false positive rate and high coverage of undetected mutants, while low execution time ensures the technique is efficient). Thanks to its high precision of 81%, MutationBERT saves 66% of the time spent by prior work to verify live mutants, and improves precision, recall, and F1 score in both same project and cross project settings. This 66% time savings includes model inference time, with the cost of training the model being a one-time cost that is amortized over many uses.

We extend prior evaluation of predictive mutation tools to focus on a practical setting, where developers are not shown false positives. We also measure performance on non-trivial mutants, which are more important to classify correctly; trivial mutants are detected by every test and are especially uninteresting for developers. Using MutationBERT takes 33% of total mutation testing time even when verifying all predicted live mutants, while also improving performance on non-trivial mutants over prior approaches. In summary, the contributions of this chapter are:

[1] Work that appeared in FSE 2023 [61]

```
public RegularTimePeriod next() {
  Hour result;
-  if (this.hour != LAST_HOUR_IN_DAY) {
+  if (this.hour > LAST_HOUR_IN_DAY) {
    result = new Hour(this.hour + 1, this.day);
  }
  ...
}

public void testNext() {
  Hour h = new Hour(1, 12, 23, 2000);
  h = (Hour) h.next();
  assertEquals(2000, h.getYear());
  ...
}
```

(a) Motivating example

```
<CLS>
public RegularTimePeriod next() {
  Hour result;
  if (this.hour <BEFORE> != <AFTER> > <ENDDIFF>
    LAST_HOUR_IN_DAY) {
    result = new Hour(this.hour + 1, this.day);
  }
  ...
}
<SEP>
public void testNext() {
  Hour h = new Hour(1, 12, 23, 2000);
  h = (Hour) h.next();
  assertEquals(2000, h.getYear());
  ...
}
```

(b) Model encoding of example

Figure 4.1: A snippet of code from the popular JFreeChart Java project, where a mutation changing != to > is applied (Figure 4.1a). The provided test fails to detect this mutant. Figure 4.1b shows how we encode this mutant in our approach. Newly added special tokens are highlighted.

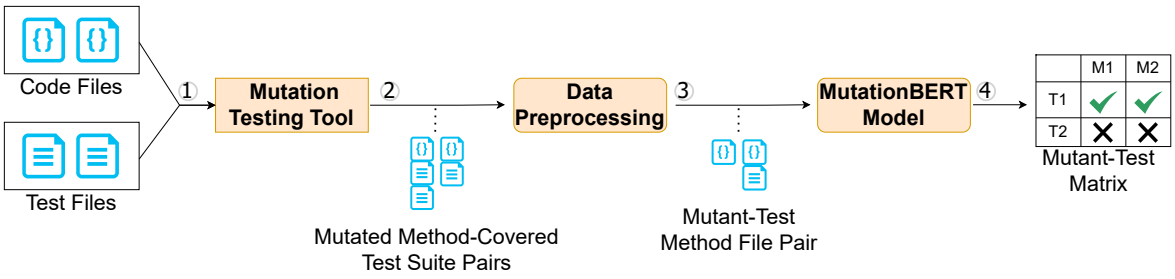


Figure 4.2: An overview of MutationBERT's workflow. Step ① provides source and test files to a mutation testing tool. In Step ②, the mutation tool generates mutants and correspondng covering tests, which are preprocessed, tokenized, and formatted. In Step ③, MutationBERT takes these inputs to produce (Step ④) the full mutant-test matrix.

- An empirical evaluation of predictive mutation testing tools, measuring both inference time and the runtime cost savings.
- MutationBERT, the first predictive mutation testing model to incorporate source and test code context. MutationBERT can predict entire mutant-test matrices along with whether mutants are detected or not by test suites.
- An analysis of the design decisions, including an examination of alternative input representations that leverage both source and test method context.

4.1 Contextual Predictive Mutation Testing

Figure 4.2 shows the MutationBERT workflow. Our workflow takes a project and test suite as input, and uses a given source-level mutation testing tool (step ①) generates a set of mutants and

tests that cover them (step ②). Most mutation testing tools provide coverage out of the box, as a way to prune uncovered mutants, which will always be undetected. We encode the method/test pairs in an input representation (step ③, Section 4.1.1), to be passed as input to our trained model (step ④, Section 4.1.2). The model predicts whether the test will detect or fail to detect the mutant (step ⑤). Over all mutant-test pairs, these predictions comprise the mutant-test matrix for the program. This output can be optionally post-processed to aggregate predictions across the whole test suite. This produces for the user a set of mutants likely undetected by the test suite; these can be inspected directly, or ranked by existing mutant prioritization algorithms [71, 109**?**]. As the developer adds tests, more interesting mutants are identified, leading to better test suites over time.

As an illustrative example, consider Figure 4.1a, which shows a (simplified) code and test snippet from JFreeChart.[2] The `next()` method returns the next hour for the class under test: `RegularTimePeriod`. The `testNext` method checks that it works correctly for 23:00 on December 1st, 2000. Although this test method may look comprehensive, note that it does not fail if we change the `!=` operator to `>` on line 3. A better test suite would include another method that includes a time that is not the last hour of a day, which would correctly fail on the mutated code. We will refer to this example throughout subsequent sections to clarify our contribution.

4.1.1 Input Representation

Our goal is to train a model that predicts whether a given test will detect a given mutant. Concretely, a mutant is a typically small modification to a typically much larger code file. Prior efforts to represent code changes for the purpose of ML, fall into three main categories: defining a set of features related to the modification [74, 146] representing the modification with a graph [90, 134, 145] or representing the "before" and "after" of the modification with multiple embeddings [123].

For earlier PMT models [74, 146] that did not use pretrained transformers, defining a set of features and aggregating them into a single vector made sense. However, to leverage the gains from using a pretrained model like CodeBERT [43], we need to represent our inputs in the same way as the pretrained model, making the feature-based approach unviable. Following best practices in pretrained transformers, we use the same input embeddings for encoding the mutated code and the tests.

Thus, we represent each mutant-test pair as a token level diff to MutationBERT, using the special tokens `<BEFORE>`, `<AFTER>` and `<ENDDIFF>`. For example, if the line `...if a == b:...` is changed to `...if a != b:...`, we encode it in the following manner: `...if a <BEFORE> == <AFTER> != <ENDDIFF> b:...`. This encode diffs compactly, while preserving original code structure.

Figure 4.1b shows how our model encodes the motivating example. We provide the model with the source method encoded as a token-level diff, followed by the test method. Our model then outputs whether such a mutant is detected or undetected. We follow CodeBERT [43] in their use of special tokens `<CLS>` and `<SEP>`. CodeBERT uses `<CLS>` and `<SEP>` to denote code and natural language input, using `<CLS>` token for downstream classification tasks (we discuss

[2] `https://github.com/jfree/jfreechart`

this in more detail in Section 4.1.2). Similarly, we separate code and test with the special <SEP> token. We take the hidden representation of the <CLS> token as the vector which we train the model to classify whether this mutant is detected or not.

4.1.2 Model

Our model can predict either the entire mutant-test matrix for a project, or whether a single mutant is detected by an entire test suite. Our model is a pretrained CodeBERT model fine-tuned to the mutation testing task, with a novel input representation. CodeBERT [43] is a pretrained model that leverages the transformer architecture [131]. It was trained to predict *masked* tokens (code or natural language tokens replaced with <MASK>) for both source code and natural language. CodeBERT uses special <CLS> and <SEP> tokens to denote code and natural language, using the <CLS> token for classification in downstream tasks. CodeBERT was pretrained on a corpus of 6.4 million functions across seven different programming languages; large pretrained models like CodeBERT are applicable to a variety of downstream tasks ranging from code completion [43], to merge conflict resolution [123], and code summarization [8]. To the best of our knowledge, we are the first to leverage pretrained models for the task of predictive mutation testing.

We formulate mutation analysis as a binary classification task to CodeBERT. We provide CodeBERT with both the source method encoded as a token level diff and the test method (Section 4.1.1). After feeding the input to CodeBERT, we pass the encoding of the <CLS> token through a linear layer, which is then used to make the final classification. The model is called for each mutant-test pair to construct the entire mutant-test matrix.

We use the probability output of the model to aggregate predictions across each mutant's set of covered tests, and consider a mutant to be "detected" if the confidence of the model on at least *one* of the tests is greater than 0.25:

$$\texttt{pred}_{M,T} = \begin{cases} \text{“detected”} & \text{if } \max_{t \in T} \text{MutationBERT}(M, t) > 0.25 \\ \text{“undetected”} & \text{otherwise} \end{cases} \tag{4.1}$$

where M corresponds to the mutant and T corresponds to the set of tests that cover the mutant. We chose 0.25 as our confidence threshold, as it was able to reduce the number of false positives when evaluated on our validation dataset, with a precision of 0.76, while not reducing the overall *F1* score of 0.80.[3]

4.2 Experimental Setup

We compare MutationBERT with Seshat [74], the current state-of-the-art model for PMT, using the dataset from that paper. We also consider different input aggregation approaches in published work [61]. Our evaluation extends prior work [74, 146] by considering the practical setting, where developers are not shown false positives and adding an additional experiment measuring performance on hard-to-detect mutants (mutants with a small proportion of tests detecting them).

[3]Full details can be found in published work [61]

We ask the following research questions:

RQ1: Effectiveness: How well does MutationBERT perform in a *same project* setting? In a *same project* setting, a PMT model is trained on previous versions of a project, and then used to predict test matrices, unkilled mutants, or mutation scores for subsequent versions. We compare MutationBERT to Seshat on a within-project task, evaluating the models' correctness when predicting test-mutant matrices and over the test suite- level aggregation. We ask this question because same project settings align with developers using predictive mutation testing techniques as their project evolves (paying the one time cost at the beginning).

RQ2: Generality: How well does MutationBERT perform in a *cross project* setting? In a *cross project* setting, a PMT model is trained using data from one project and then used to predict test-mutant behavior for a different project. This is much more difficult than the same project setting, but could be especially applicable when starting a new project, for example. We compare MutationBERT to Seshat on the cross-project task using the same metrics as the *same project* task. We ask this question because cross-project settings measure how whether a developer could use our model out of the box with no additional training.

RQ3: Design Decisions: How do different input representations and aggregation approaches affect our final model? We analyze and compare several input representations as well as aggregation approaches to validate the design decisions underlying MutationBERT. We ask this question to understand the impact of our design decisions (both input representation and aggregation) on the final model's performance.

RQ4: Qualitative Analysis: What are causes of MutationBERT mispredictions? We manually examine 100 cases where our model misclassifies a mutant as detected or undetected to identify common reasons for failures and better understand limitations. We ask this question to better understand cases where our approach fails and provide insights for potential future improvement.

RQ5: Efficiency: How efficient is MutationBERT compared to prior work, and regular mutation testing? We address how MutationBERT compares to Seshat, and characterize the performance improvement it provides over regular mutation testing. We ask this question to quantify how much time our approach saves, which is the end goal of PMT techniques.

RQ6: Mutant Importance: How effective is MutationBERT at predicting difficult-to-detect mutants? We address how MutationBERT compares to Seshat with regards to how many tests detect a mutant, a proxy for mutant difficulty. We ask this question to understand how our approach performs with more difficult mutants, which are also important to classify correctly.

4.2.1 Baseline

We compare against the Seshat baseline [74]. Seshat is a state-of-the-art model for mutation testing, which has been shown to outperform PMT [146] by 0.14 to 0.45 *F1* score depending on project. Similar to our model, Seshat has no overhead in static or dynamic analysis, operating entirely on source level features, unlike the prior model PMT, which requires both static and dynamic analysis to run. However, unlike our model, Seshat operates over a set of features: the

Table 4.1: Our dataset comprising of 6 Defects4J 2.0 projects.

Project	**Date**	**LOC**	**#tests**
commons-lang	2013-07-26	21,788	2,291
jfreechart	2010-02-09	96,382	2,193
gson	2017-05-31	7,826	1,029
commons-cli	2010-06-17	2,497	354
jackson-core	2019-01-06	25,218	573
commons-csv	2017-12-11	1,619	290

Table 4.2: Tests, mutants and mutant-test pairs (pairs) for both same project and cross project settings, across training (train), validation (val), and test (test) sets. Note that mutant-test pairs only include tests that cover a given mutation.

	Split	**#tests**	**#mutants**	**#pairs**
Same Project	train	6,124	68,702	1,522,924
	val	5,644	8,688	197,527
	test	5,637	8,648	195,140
Cross Project	train	4,725	79,128	1,460,344
	val	1,171	5,427	402,296
	test	261	1,040	42,687

source method name, the test method name, the mutated line before and after, and a one-hot encoding of the mutation operator. Seshat first encodes the source and test method names with a bidirectional GRU. It then concatenates the resulting embeddings with a one-hot encoding of the mutation operator to classify the mutant as detected or undetected by the test.

Like our model, Seshat outputs a confidence score for each mutant-test pair, which we aggregate to predict whether the mutant is detected or not by the entire test suite. We aggregate Seshat's predictions across each mutant's set of covered tests by comparing confidence to a threshold. We set this threshold to 0.10, which in our experiments produced the highest *F1* score for Seshat in validation (Seshat does not mention a threshold in their paper, so we perform the same optimization as we did for MutationBERT).[4] We thus aggregate as follows:

$$\texttt{pred}_{M,T} = \begin{cases} \text{“detected”} & \text{if } \max\limits_{t \in T} \text{Seshat}(M, t) > 0.10 \\ \text{“undetected”} & \text{otherwise} \end{cases} \tag{4.2}$$

where `M` corresponds to the mutant and `T` corresponds to the set of tests that cover the mutant.

4.2.2 Dataset

We reuse the dataset released with the Seshat experiments [74]. This dataset consists of a full mutation analysis in Major [67] of six large scale Java projects, with extensive testing, across multiple versions, taken from Defects4J v2.0.0 (statistics shown in Table 4.1). This dataset considers only mutants that are actually covered by some test, since uncovered mutants cannot be detected by a given test suite (and can be discarded with a simple coverage heuristic).

Note that the Seshat evaluation [74] analyzed the cross-version setting in detail, training models on previous versions of programs to predict matrices for subsequent versions. The models remain effective across versions many years apart. This is likely a function of the fact that code (and mutation behavior) is quite stable over time, as shown in the dataset description in Kim et al. [74].

Thus, in the interest of space and computational effort, we restrict our attention to single versions per project for all RQs. We select the latest versions of the six projects in Defects4J 2.0 and perform an 80-10-10 split between train, validation and test sets. In the same project setting, we split by mutant-test suite pair. This is in contrast to the prior evaluation, that is, mutant-test pairs from the *same* test suite must be part of the same subset. Practically, our envisioned application does not include a situation where a PMT model could be trained on data corresponding to whether half the tests in a given test suite detect a given mutant, and then used to predict the behavior of the other half. This explains why we reran Seshat (and why our numbers may not match those in the original paper). For the cross project setting, we split by project, where each project consists of a set of mutant-test suite pairs. We use the exact same splits for our model and for Seshat. Table 4.2 shows statistics about our same project and cross project splits.

4.2.3 Preprocessing and Training

We use the pretrained RoBERTa tokenizer (BPE tokenizer [120]) with vocabulary size of 50,000 tokens for all programming languages that are provided with CodeBERT. We fine-tune CodeBERT with context window size of 1024 tokens, and thus only provide MutationBERT the first 1024 tokens of the code and test combinations. Such cases account for 14.6% of all mutant test pairs.

We follow the same steps that Kim et al. [74] took to train Seshat. We train Seshat for 10 epochs, with a batch size of 512, and learning rate of `3e-3`. We train MutationBERT for eight epochs with learning rate of `1e-5` and batch size of 64. We use a weighted loss function according to the distribution of detected and undetected mutant-test pairs. We use a linear warm up to 1000 steps, followed by a cosine annealing decay, in accordance with best practices for fine-tuning transformers [112]. Both models' loss functions converge using these settings. We fine-tuned our model on an Nvidia GeForce RTX 3080 for one week for a total of 115k steps.

4.2.4 Metrics and Settings

One way to use models for predictive mutation testing is to compute mutant-test matrices, which predict, for each mutant, whether each test passes or fails. In general, most tests pass on most

[4]Full details on this comparison can be found in published work [61]

mutants. That is, a test detecting a mutant is the minority class. In this setting, model *precision* refers to how accurately mutants are identified as detected, while *recall* refers to the proportion of detected mutants labeled correctly. In the mutant-test matrix setting 72% of mutant-test pairs are undetected. We care that our model is able to accurately predict the remaining 28% of detected mutants; the goal is to identify the few tests that detect each mutant.

Another way to use these models is to predict whether an entire test suite detects a particular mutant. Here, the majority class is detected mutants; 61% of mutants are detected. The core goal here is to accurately identify the undetected mutants, to guide developers to improve test suites. Therefore, we define precision and recall differently than in the mutant-test matrix setting. In the test suite setting, model *precision* refers to how accurately mutants are identified as *undetected*, while recall refers to the proportion of *undetected* mutants that are classified correctly. *Precision* is thus important in understanding the potential cost of a PMT model in terms of time needed to either actual run the test suite to confirm its predictions, or time wasted by a developer inspecting an ultimately uninteresting mutant. *Recall* is also important to overall model usefulness: if a model misses a large number of undetected mutants, key gaps in test suite quality could remain.

We report precision, recall and F1 score (which balances the two) for all models in the first three research questions. For RQ1 (same project) and RQ2 (cross project), we evaluate performance both on the base test set (195,140 mutant-test pairs). For efficacy of prediction over the entire test suite, we evaluate MutationBERT on the same dataset, aggregated at the test suite level (8648 test suites).

For RQ3, we evaluate different aggregation thresholds and input representation choices on the validation set consisting of 120,710 mutants, again reporting precision, recall, and F1 scores; we evaluate both mutant-test predictions and mutant-test suite predictions. Due to compute constraints associated with a larger context window, we use the 512 token context window to evaluate different thresholds and input representations.

For RQ4, to ensure a representative sample of misclassifications, we randomly select 100 examples where our model misclassifies a mutant as being detected or undetected. We manually examine each example and try to understand the cause of the misprediction. Finally, we bucket these mispredictions in a series of categories and discuss these in detail. We do this to inform a general assay of the limitations of our technique; we do not make strong claims about the generalizability of this qualitative assessment.

For RQ5, we run 1000 iterations of Seshat and MutationBERT, with a batch size of one, on a workstation with an Nvidia GeForce RTX 3080 GPU, with 100 warm up iterations. We report the average time taken over these 1000 iterations as the inference time for each model. To compute comparative time and speedups against regular mutation testing, we use numbers from previous work [74] in conjunction with our inference time numbers.

For RQ6, we report accuracy of Seshat and MutationBERT with respect to percentage of tests that kill a mutant. The goal is to measure whether MutationBERT is only correctly classifying ”easy” to detect or ”trivial” mutants where the majority of tests detect the given mutant or whether MutationBERT is capable of correctly classifying mutants that are more difficult to detect.

4.3 Limitations and Threats

Limitations: MutationBERT depends on GPU availability to efficiently make predictions. On a CPU, MutationBERT takes 84 milliseconds per prediction, or 12 mutant-test pairs per second (a far cry from the 29 mutant-test pairs per second on a GPU). These times are still significantly faster than running the tests themselves, which on average takes 80 times longer than inference of MutationBERT on GPU. Note that both these CPU and GPU times are theoretical worst cases, since these times were computed using a batch size of one. Many current CI pipelines are largely CPU-based, potentially compromising practical utility. However, cloud providers increasingly provide GPU access; recently, GitHub actions announced plans to do the same for CI.[5] Indeed, GPUs are becoming more broadly accessible, including via idle GPU time or services like Google Colab. Future testing approaches are thus increasingly realistic to deploy in practice.
Threats to Validity: The main internal threat to validity is that there might be defects in our implementation of MutationBERT. We used widely available and popular libraries such as PyTorch and Pandas for managing data and building the model to help mitigate this threat. We release our models and implementation for inspection and extension by others.

The main external threat to validity is that our dataset of mutants and tests might not generalize to all projects. We reused the data produced by prior work on a large dataset (Defects4J) that has been used and validated in many other studies in software engineering. Since this dataset is sourced from multiple different projects, the results are more likely to generalize.

Finally, threats to construct validity lie primarily in our evaluation metrics. We report widely used metrics in machine learning, i.e., precision, recall and F1 score. While precision and recall correlate with false positive and false negative rates respectively, they do not perfectly capture end user experience (for example, a model with high precision but low recall might be more useful than a model with lower precision but higher recall if developers are strongly sensitive to false positives).

4.4 Results and Analysis

We report results for all five RQs, and discuss their implications.

4.4.1 RQ1: Same Project Performance

Table 4.3 shows the results of MutationBERT and Seshat on the test set for the *same project* setting. MutationBERT outperforms Seshat across all metrics: MutationBERT's *F1* score is 0.75, compared to Seshat's 0.67. Interestingly, MutationBERT and Seshat have similar precision (0.66 for Seshat vs 0.72 for MutationBERT); the models report similar numbers of false positives (cases where the models misclassify a test as detecting a mutant). However, MutationBERT has higher recall (0.77, versus 0.68), meaning that MutationBERT is more likely to correctly identify cases where a test detects a mutant.

When the predictions are aggregated into test suite level predictions (right-hand columns), recall that undetected mutants are the minority class, flipping the meaning of precision and recall

[5] `https://github.com/github/roadmap/issues/505`

Table 4.3: Comparison between Seshat and MutationBERT on both same project and cross project settings in terms of precision, recall and F1 score. In both same project and cross project settings, MutationBERT outperforms Seshat across all metrics, with an *F1* score difference of 12% on the same project setting and *F1* score difference of 28% on the cross project setting. The center columns show results in predicting whether a test will detect a particular mutant, relevant to constructing the overall mutant-test matrix.

Setting	**Model**	**Mutant-Test Matrix**			**Test Suite**		
		Precision	**Recall**	**F1**	**Precision**	**Recall**	**F1**
Same Project	Seshat	0.66	0.68	0.67	0.56	**0.82**	0.67
	MutationBERT	**0.72**	**0.77**	**0.75**	**0.81**	0.78	**0.79**
Cross Project	Seshat	0.58	0.29	0.38	0.24	0.39	0.30
	MutationBERT	**0.68**	**0.37**	**0.48**	**0.52**	**0.65**	**0.58**

(Section 4.2.4). Seshat and MutationBERT both find similar numbers of undetected mutants, but MutationBERT has much higher precision, 0.81, compared to Seshat's 0.56. False positives are costly, as they cost developers valuable time examining mutants that are in reality detected by their test suite.

Another way of viewing these results is in terms of the difference between the mutation score estimated by a predictive mutation model, and the actual mutation score. Recall that mutation score is the true ratio of detected mutants to total mutants; empirically, mutation score provides a better measure of test adequacy than code coverage [68, 107] and thus is useful (albeit usually expensive) to compute. The gold mutation score (true mutation score) on our test set is 0.59. Seshat estimates a mutation score of 0.40 over the entire dataset, an error of 0.19. MutationBERT computes a mutation score of 0.61, a difference of only 0.02 from the true answer. MutationBERT thus has much lower error in estimating mutation score on this dataset as compared to Seshat.

4.4.2 RQ2: Cross Project Performance

Table 4.3 also shows the *cross project* setting (bottom rows), where a model is trained on one set of projects and evaluated on another. Again, MutationBERT outperforms Seshat (0.68 precision and 0.37 recall for MutationBERT and 0.58 precision and 0.29 recall for Seshat). That said, in the mutant-test predictions, both precision and recall drop significantly for both approaches; this suggests that training data containing project-specific vocabulary and methods contribute substantially to the same project performance. This is consistent with other results showing that projects have distinct vocabulary and style, making cross project prediction difficult for many tasks [9, 54]. Precision continues to be quite a bit higher than recall in the cross project setting, for both models.

At the test suite level, we find that MutationBERT outperforms Seshat on all metrics. Precision is very low for both tools; Seshat and MutationBERT both misclassify a significant proportion of undetected mutants, however MutationBERT has a significantly higher precision. Recall is also low in the cross project setting, at 0.39 for Seshat and 0.65 for MutationBERT. However,

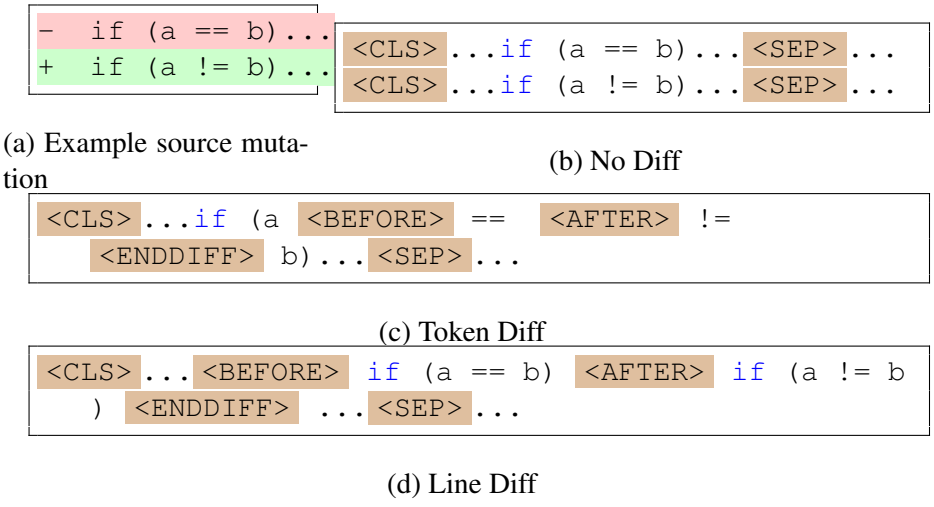


Figure 4.3: Input representations for encoding mutations applied to source code. Each subfigure shows a different input representation on the same example of changing == to !=. Token diff and line diff were the best performing input representations and we chose to use token diff as the final input representation in MutationBERT.

this indicates that in a cross project setting MutationBERT is capable of finding more undetected mutants than Seshat.

On the cross project test set, the gold mutation score is 0.77. Seshat differs from this value significantly, with a mutation score of 0.63 (error of 0.14). MutationBERT is much closer, predicting a mutation score of 0.72 (error of 0.05).

4.4.3 RQ3: Input Representations and Aggregation Approaches

We proposed a new input representation for the mutation prediction problem. Here, we describe several alternatives that we then experimentally evaluate. We also describe alternative aggregation approaches. Then, we evaluate these alternatives (all on the validation set) to motivate the input representation and aggregation approaches in our final model.

Input Representations

We outline various input representations that incorporate source and test context for our model. For all input representations, we separate method code and test code with a ¡CLS¿ token, which we use for classification.

No Diff (Binary Task): Our simplest approach is to directly apply the mutation and feed the model both the mutated version of the code and unmutated version of the code. For example, when changing == to != in `...if a == b:...` we feed the model both `...if a == b:...` and `...if a != b:...` (Figure 4.3b).

Since we have likelihood scores for both the mutated and unmutated versions of the code, we try two modes of evaluation. Our first mode feeds the model the mutated code, and takes its prediction. Our second mode feeds the model both the mutated code and unmutated code and

obtains its probability of being detected. Then it subtracts these two probabilities from each other (since we know the first datapoint is always undetected), and compares this difference against a dynamically set threshold. We try all thresholds between 0.01 and 0.99 in increments of 0.01 on the validation set, and select the best performing threshold.

Token Level Diff: We represent each mutation as a token level diff. For example if a line `...if a == b:...` is changed to `...if a != b:...`, we encode it in the following manner: `...if a <BEFORE> == <AFTER> != <ENDDIFF> b:...` (Figure 4.3c). This allows for the most compact footprint in encoding the diffs, allowing our model to learn how certain diffs coupled with the surrounding code and test are correlated with a mutant being detected or not detected.

Line Level Diff: For line level diffs, we represent diffs in terms of change to source lines. This input representation is similar to token level diff. In our example, we encode the mutation as `...<BEFORE> if a == b: <AFTER> if a != b: <ENDDIFF> ...` (Figure 4.3d). We hypothesize that this might perform better than token diff, as CodeBERT was pretrained for tasks such as next line prediction.

Aggregation Approaches

We outline aggregation approaches that we tried for our test matrix model. Practically, this aggregation holds value, as undetected mutants (mutants not detected by the entire test suite) are ones of interest to developers, as they indicate testing inadequacy. Specifically, in order to use such a model, aggregate predictions need to be accurate, otherwise undetected mutants will be identified incorrectly.

Threshold Aggregation: We aggregate the predictions of both predictive mutation testing models by using various probability thresholds (0.1, 0.25, 0.5, 0.75 and 0.9). Specifically, we only label a test as detecting a mutant if the model predicts the test detects the mutant with probability above the defined threshold. We vary thresholds to observe their effect on precision, recall, and F1 score.

Learned Aggregation: We also tried learning an aggregation based off of the embeddings of the `<CLS>` token after CodeBERT encoding. We use a transformer with three layers to take these embeddings and aggregate them. We then use a linear layer to classify based off of this learned aggregate embedding whether the test suite detects or fails to detect the mutant. We evaluate this learned aggregation both using a weighted loss function (according to the data distribution) and using a normal loss function.

Experimental Results

We evaluate input representations on our validation set for Defects4J 2.0. The data distribution is 72% undetected and 28% detected for test matrices. The *No Diff* model requires two examples per mutant, making an even more unbalanced distribution (86% undetected, 14% detected). Therefore, in training these models, we use a weighted loss function that penalizes misclassifications of detected mutants more than undetected mutants. The weights are different for the *Token Diff* and *Line Diff* models and the *No Diff* model.

Table 4.4: Precision, recall and *F1* scores of all models at predicting the mutant-test matrix on the validation set. Token diff and line diff are the best performing models, with an F1 score of 0.78.

Model	Precision	Recall	F1
Seshat	0.73	0.75	0.74
Token Diff	**0.79**	**0.77**	**0.78**
Line Diff	**0.79**	**0.77**	**0.78**
No Diff (Normal)	0.74	0.72	0.73
No Diff (Threshold - 0.01)	0.73	0.72	0.73

Table 4.4 compares our novel input representations against the baseline Seshat model. *Token Diff* and *Line Diff* perform almost identically, with approximately a 4% improvement in F1 score over baseline (we use the token diff model for our other results). Somewhat surprisingly, when the diff is not explicitly specified (in the *No Diff* models), the model fails to reason about how code relates to tests passing or failing This is further supported by the thresholding (in the *No Diff* models) having no effect on validation F1 score (regardless of what the threshold is from 0.01 to 0.99). We hypothesize that knowing the mutation applied is a key piece of context for accurate predictions. Both our token and line diff models have tokens that specify the start and end of the applied operator.

We similarly evaluate aggregation strategies on the validation set, at the test suite level (the goal of the aggregation strategies is to predict over test suites). Table 4.5 shows results of all aggregation strategies we tried on the validation set.

We find that even with the small change in F1 score between the two models for test matrix prediction, there is significant change in F1 score when it is aggregated at the test suite level. This is due to the compounding effect of errors, as an error in any one of the tests in the test matrix can cause the whole suite to be labeled incorrectly, making even a small difference in F1 score equate to large differences in the aggregated matrix.

To select thresholds, we use the validation set and the F1 score followed by precision. Precision is more important than recall here, because the cost of a false positive is high. Specifically, a false positive means that a developer will see a mutant that is supposed to indicate test inadequacy when in reality their tests are adequate. We find that the best threshold for Seshat is 0.10 and the best threshold for MutationBERT is 0.25.

4.4.4 RQ4: Tool Misclassifications

To understand our model's limitations, we examined 100 randomly sampled examples of MutationBERT misclassifications from our validation set. We categorize causes of failures in Table 4.6. Upon inspection, we classified each example into two high-level buckets: *Not enough context* and *Missed clue*. *Not enough context* refers to cases where the model was missing context that even a human would need to classify the case correctly. The large majority of our examples (71/100) fell under this bucket. The second category consists of *Missed clue*s, where the model missed some crucial clue to mutant behavior (29/100).

Table 4.5: Threshold and aggregation approaches, predicting test suites on the validation set. The best threshold for Seshat is 0.10; for MutationBERT, 0.25. We find that the transformer aggregation approaches have lower precision than the selected threshold approach, meaning more false positives.

Model	**Threshold**	**Precision**	**Recall**	**F1**
Seshat	0.10	0.57	0.83	0.67
	0.25	0.56	0.85	0.67
	0.50	0.48	0.92	0.66
	0.75	0.52	0.87	0.65
	0.90	0.51	0.89	0.65
MutationBERT	0.10	0.76	0.84	0.80
	0.25	0.76	0.84	0.80
	0.50	0.75	0.86	0.80
	0.75	0.74	0.87	0.80
	0.90	0.73	0.88	0.80
trans (weighted)	N/A	0.75	0.85	0.80
trans (unweighted)	N/A	0.75	0.85	0.80

Table 4.6: Reasons MutationBERT incorrectly classifies mutants. In 71/100 cases, MutationBERT lacks sufficient context, while in the remaining 29/100 cases MutationBERT misses a contextual clue.

Category	**Case**	**Count**
Not enough context	Helper test method	44
	Method	24
	Class	3
Missed clue	Code	22
	Method name	7

We were able to subdivide the high-level buckets into common subcategories. For *Not enough context* these are *Helper test method*, *Method* and *Class*. *Helper test method* refers to cases where the test method consists primarily of invocations to another method. One example is as follows:

```
public void testJava2DToValue() {
  checkPointsToValue(edge, plotArea);
  this.axis.setRange(0.5, 10);
  checkPointsToValue(edge, plotArea);
  ...
}
```

Test method `testJava2DToValue` invokes helper method `checkPointsToValue` multiple times. Without the helper method code, MutationBERT lacks the context (or even knowledge of relevant test assertions) to make an accurate prediction on any mutant.

The *Method* category refers to the model lacking necessary source context. For example:

```
public <T> TypeAdapter<T> create(...)

public void testDeserializeNullField() throws IOException {
  Truck truck = truckAdapter.fromJson(...);
  ...
}
```

This example shows a test that invokes the `fromJson` method, which then invokes `create`. Without the code for `fromJson`, MutationBERT cannot reason about how a mutant in `create` would affect a test calling `fromJson`.

Finally *Class* refers to cases where the constructor of a class is mutated, but the test invokes a subclass and thus is missing the subclass constructor context. The following example shows this:

```
public StrokeMap()

public void testCloning() {
  PiePlot p1 = new PiePlot();
  ...
}
```

In this example, `testCloning` is invoking the constructor of `PiePlot`, which is a subclass of `StrokeMap`. Without seeing the constructor of `PiePlot`, MutationBERT cannot understand how mutants to the `StrokeMap` constructor affect the test.

Missed clue is divided into *Code* and *Method name*. *Code* refers to cases where the model missed a context clue in the source code that indicated that the mutant was detected. For example:

```
1  public boolean hasNext() throws IOException {
2    ...
3  - return p != PEEKED_END_OBJECT
4  -   && p != PEEKED_END_ARRAY;
5  + return true && p != PEEKED_END_ARRAY;
6  }
7
8  public void testDoubleArrayDeserialization() {
9    double[] values = gson.fromJson(...)
10   assertEquals(0.0, values[0]);
11   ...
12 }
13
```

In this example, the mutant on line 3, replaces the object check with true, but the test is only for arrays. Thus, the mutant will not be detected by the provided test, since the object check is not being tested. MutationBERT misses the correlation between the object check and the test asserts all looking at arrays.

Finally, *Method name* refers to cases where the model fails to detect an important context clue in the method name. For example:

```
1  public BufferedImage createBufferedImage(...,
       ChartRenderingInfo info) {
2    ...
3  - if (info != null) {
4  + if (true) {
5      info.setRenderingSource(...);
6    }
7  }
8
9  public void testDrawWithNullInfo()
10
```

This example shows a mutant that replaces a null check on `info` with `true`. Since the test is a case where `info` is null, on the mutated code, there will be a null pointer dereference. Thus, a `NullPointerException` will be thrown and the mutant will be killed. MutationBERT fails to see the correlation between the test name and the mutant applied.

4.4.5 RQ5: Efficiency

Finally, we discuss the efficiency and performance benefits of MutationBERT as compared to Major or Seshat. Table 4.7 shows time to run each tool, including Major, for all mutants in a project (center column), and time to run including a confirmatory check for the predictive techniques (right-hand columns).

Seshat and MutationBERT have comparable inference time in our experiments: 34 ms for MutationBERT and 17 ms for Seshat. In terms of practical impact on a user interested in per-mutant prediction, the difference between 17 and 34 ms is negligible. Meanwhile, as Table 4.7

Table 4.7: Time to run Major, MutationBERT, and Seshat, over all mutants (center columns), or incorporating a confirmation check before presenting unkilled mutants to the user (right-hand columns).

		No Checking		Checking	
Project	**Major (s)**	**Us (s)**	**Seshat (s)**	**Us (s)**	**Seshat (s)**
commons-lang	12,924	748	374	3324	5767
jfreechart	64,719	1424	712	18458	23838
gson	16,738	150	75	6136	8611
commons-cli	1,290	53	26	542	841
jackson-core	113,343	809	405	33035	52231
commons-csv	5,289	36	18	1458	2550

shows, the time required to compute a full mutation score for a given project is the same order of magnitude (10s of minutes), while both an order-of-magnitude faster than Major.

However, despite being slower than Seshat on a per-prediction basis, MutationBERT still offers significant computational savings for the end-user aiming to improve a test suite (the original goal of mutation testing, and consistent with its use at companies like Google and Meta). In this setting, the user receives a list of undetected mutants to inspect and use to create new tests. A practical application for predictive mutation testing should include a *check* of each predicted-undetected mutant before presenting the list to the developer to filter incorrect predictions; this ensures that the tool is presenting truly actionable information and saves the developer time and frustration in confirming the tool's results. The right-hand-side of Table 4.7 shows that because MutationBERT has higher precision than Seshat (and similar recall), its predictions can be verified and thus put to use by the developer much more quickly.

4.4.6 RQ6: Mutant Importance

Figure 4.4 shows model accuracy of both Seshat and MutationBERT with respect to percentage of detecting tests in a given mutant's test suite. Mutants with a high proportion of detecting tests are likely to be trivial, while mutants with few detecting tests are more likely to be interesting. We compare MutationBERT to Seshat in detecting trivial vs hard to detect mutants by reporting model accuracy as a function of percentage of detecting tests. Mutants that are killed by all tests are trivial, and we hypothesize they are easier for models to detect, while mutants with fewer detecting tests are more likely to be interesting and more difficult for models to detect.

As expected, both approaches are less accurate at detecting mutants that fail fewer tests. Importantly, however, MutationBERT outperforms Seshat considerably on harder-to-detect mutants (those failing 1%-20% of the test suite), by 30%. Although Seshat is slightly more accurate at classifying mutants that fail no tests at all (0.82 accuracy vs. 0.78), MutationBERT's *overall* accuracy is higher, by 17%. Overall, MutationBERT is more accurate than prior work in predicting mutant behavior, especially the hard-to-detect cases.

(a) Seshat

(b) MutationBERT

Figure 4.4: Accuracy vs. percentage of killing mutants for Seshat and MutationBERT

4.5 Discussion

MutationBERT illustrates the importance of incorporating domain knowledge, when applying language models to software testing tasks. Prior work, missed important context in the test method body (only considering the test method name), such as the values the method under test is invoked with along with the properties being checked by asserts. Without this context, even a human would not be able to predict whether a test will detect a mutant, thus existing models are inherently limited in how well they can perform. Additionally, we apply our insight from CAT-LM [116] and fine tune with both the mutated source method and test method so the model learns the joint relationship between these connected methods. These two insights equate to increased precision, meaning significantly fewer false positives for developers.

Additionally, MutationBERT is *practically* useful for both of the core end user tasks in mutation testing: 1) as a more complete measure of testing adequacy (computing mutation score) [49, 99] and 2) to identify undetected mutants that indicate potential inadequacies in existing testing efforts [109**?**].

In the classical sense, mutation testing serves to evaluate test suite quality [39, 53, 64]. Mutation score, or the proportion of detected mutants to total mutants, provides a powerful measure of how well tested, including in terms of actual oracle strength, a given piece of code is. MutationBERT drastically reduces the amount of time needed to compute mutation score, taking approximately 30 ms per mutant test pair, substantially lower than the actual cost of executing a test (and compiling mutants). The error rate of MutationBERT is also low, with MutationBERT having below a 5% error in predicting mutation score for both same and cross project settings, substantially lower than Seshat. Further note that as Table 4.7 shows, it is plausible that using MutationBERT to approximate mutation score will be faster (in our data, about twice as fast) as even approximating score by sampling as few as 10% of mutants. Sampling 10% of mutants is likely to be no more accurate than MutationBERT [49], and additionally provides *no* data on mutants not sampled, while our approach provides a good approximation of the result for all

mutants.

More recently, companies like Google [109] and Facebook [**?**] use mutation testing to pinpoint undetected mutants that reveal issues with test adequacy. MutationBERT substantially saves time here, as unlike Seshat, it still achieves over 60% accuracy in predicting hard to detect mutants. Even verifying the output of all mutants classified as undetected by MutationBERT first saves 71% of time when compared to regular mutation testing, significantly more than Seshat's 57% time savings. We note that with very high actual mutation scores (where examining unkilled mutants is most useful), the time required to discover n undetected mutants using MutationBERT is likely to be *much* better than with Seshat or traditional mutation testing.

4.6 Conclusion

In this chapter, I further leverage the relationship between mutated source code and test methods to improve existing predictive mutation testing work. We present MutationBERT, a tool for predicting both test matrices and aggregating these predictions that take as context both the mutated source method and test method. This additional context significantly improves precision over existing approaches, which only include the test method name.

We perform an extensive evaluation of our model, finding that we save 33% of Seshat's time if a developer were to verify all mutants that either model predicted as undetected. We also outperform Seshat, the state-of-the-art model by 8% *F1* score in predicting test matrices and 12% *F1* score in predicting the aggregated test suite outcome. We also achieve similar performance in the cross project setting, outperforming Seshat by 10% *F1* score in predicting test matrices and 28% *F1* score in predicting test suites.

Overall, our work illustrates the benefits of applying the joint relationship between mutated code and tests to fine-tuning predictive mutation testing models. We examine combining this insight with execution data in the next chapter.

5 Test Generation Benchmarking

In this chapter, I leverage the relationship between code and tests to construct a large scale unit test generation benchmark at the *file* level.[1] I apply software engineering insights of incorporating adequacy metrics such as mutation score and sourcing data from large scale projects to build a benchmark that more closely resembles real-world development. TESTGENEVAL is a benchmark of code test file pairs, sourced from 11 large-scale open source repositories (3,523-78,287 stars). Code and tests in TESTGENEVAL are both longer and more complex than existing benchmarks, more closely simulating test generation in large scale software settings. I use TESTGENEVAL to evaluate my unit test generation agent in a realistic setting (Chapter 6).

Popular unit test generation benchmarks remain limited in size and scope [20, 25, 116]. While existing benchmarks capture test generation abilities on simple, typically self-contained programs, there is an absence of large scale test generation benchmarks for LLMs. Other related benchmarks [63] report performance on adjacent tasks such as generating equivalence tests rather than standard unit tests, which also differs from the real-world use case. Additionally, benchmarks such as EvoSuite [45] and Pynguin [89] suffer from a lack of maintainability due to not using Docker for outdated dependencies, and are not targeted to evaluating LLMs, requiring additional effort to adapt to our tasks. Current benchmarks report pass@k, with few reporting code coverage and none reporting mutation score, despite mutation score being most correlated with real fault detection [68, 107].

Most existing benchmarks also do not measure test completion capabilities, despite many code completion benchmarks existing [84, 147]. While CAT-LM (Chapter 3) measures test completion capabilities, it measures on significantly smaller repositories, and thus is less representative of real-world development than TESTGENEVAL (see Section 5.1.3 for a more detailed comparison). Test completion can be used to add tests to an already existing unit test file and improve overall coverage. This is important for IDE auto-completion features, where given a part of a test file and the code under test, the goal is to add more tests. Test completion is also measured by many state-of-the-art software testing models [41, 96, 116, 128], yet a benchmark that measures test completion doesn't exist.

Motivated by this, we introduce TESTGENEVAL with two tasks 1) full file unit test generation and 2) test completion (see Section 3.2 for more details). Our benchmark consists of real-world projects, with each source file containing an average 1,157 lines of code (LOC) and each test file containing an average of 943 LOC. TESTGENEVAL consists of 68,647 tests from 1,210 unique code-tests file pairs. For fast iteration in low-compute settings, we also provide a smaller version of the benchmark TESTGENEVALLITE, which approximates all the metrics computed in

[1] Work that appeared in ICLR 2025 [62]

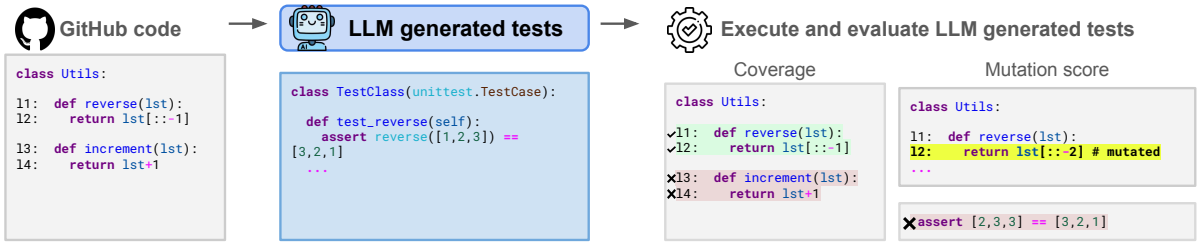


Figure 5.1: An overview of TESTGENEVAL. We start with a GitHub code file and generate a test suite with an LLM. Then we execute the generated test suite and measure the proportion of lines in the code file that are executed (code coverage). We also inject synthetic bugs into the code and measure the proportion of synthetic bugs detected by the generated test suite (mutation score).

TESTGENEVAL. TESTGENEVALLITE includes 160 code-tests file pairs, file unit test generation, and test completion tasks. It was sampled to be representative of the full TESTGENEVAL: the repositories, following the same procedure as SWEBenchLite [65].

We find that models struggle to generate high quality test suites (Section 5.2.3). The best performing model—GPT-4o—has an average coverage of 35.2% and a mutation score of 18.8%. Generating tests for large scale projects is significantly harder than generating tests for self-contained problems; this is reflected by significantly lower scores compared to existing benchmarks such as TestEval [135], where top models achieve nearly 100% line coverage. Test completion (Section 5.2.4) is significantly easier than test generation, reflected by the high pass@5 rates of the best performing models. However, models struggle to add coverage to a complete test suite, with top models adding less than 1% coverage when generating the last test for an existing file.

We perform a quantitative and qualitative analysis of all results (Section 5.3). We measure correlation between TESTGENEVAL and other popular benchmarks, the correlation between the test generation and test completion tasks, and the correlation between models for each task. We also perform an analysis of errors, and measure the effects of sampling more and context size on TESTGENEVAL performance (Section 5.3.1). We also examine cases where TESTGENEVAL can discriminate between highly performing models (Section 5.3.2). In short, the contributions for this chapter are as follows:

- TESTGENEVAL, a benchmark for partial and full test suite generation on a realistic set of 1,210 snippets in 11 repositories. We use coverage and mutation score metrics to evaluate the value of the generated test suites
- An evaluation of various prominent open and closed-source code generation models on TESTGENEVAL. We show that, for large scale repositories, models struggle to generate high coverage test suites
- Docker images and easy LLM integration allowing users to easily run code from these 11 repositories and evaluate scores on our benchmark

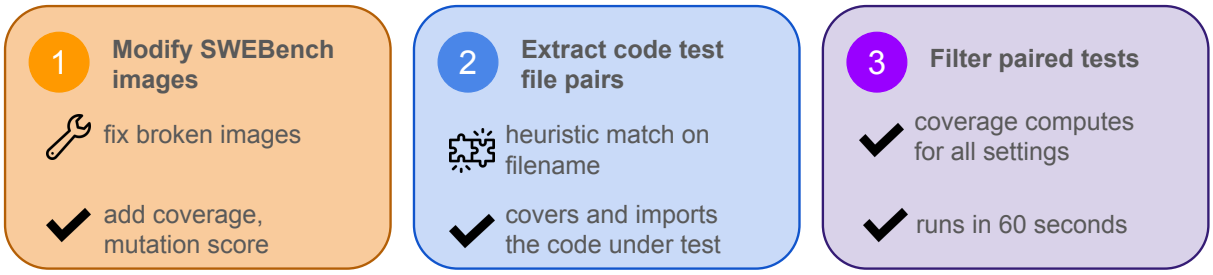


Figure 5.2: A pipeline describing the creation of TESTGENEVAL. We start with Docker images of the SWEBench dataset and instrument them with coverage and mutation score dependencies. Then we extract code test file pairs by performing a heuristic match on filename. Finally, we filter out tests that don't have coverage on the code under test and run in 60 seconds.

5.1 TESTGENEVAL

TESTGENEVAL consists of 1,210 code test file pairs from 11 large, well-maintained repositories (3,523-78,287 stars). We use these file pairs to construct two testing tasks: 1) unit test completion for the first, last and additional tests and 2) full file unit test generation. Our benchmark is easy to run and extend, as we have Docker containers for each version of each repository with coverage and mutation testing dependencies installed. For both tasks we use execution based metrics, including pass@1, pass@5 along with code coverage improvement, and mutation score improvement compared to the gold (human written) tests. Code and test files in TESTGENEVAL are long in length (on average 782 LOC per code file and 677 LOC per test file) and high coverage (median coverage of 60.4%).

5.1.1 Benchmark Construction

We construct TESTGENEVAL by adapting the SWEBench dataset to software testing tasks. SWEBench is meant to simulate real-world software development, consisting of GitHub issues in large scale open source repositories and their corresponding fixes (pull requests). We construct TESTGENEVAL from SWEBench, as the repositories in SWEBench are large scale and well maintained, ensuring that our test generation benchmarking aligns with real-world test authoring. Figure 5.2 shows the full set of steps to adapt SWEBench to TESTGENEVAL (first modifying Docker images to add testing libraries, extracting code-test file pairs and finally filtering cases where we can't compute coverage). We perform these steps to maximize replicability of TESTGENEVAL (using Docker images), while also ensuring that we can compute coverage for all data points (mapping code to tests and ensuring we can compute coverage for the code under test). We apply these exact steps to SWEBenchLite (a smaller, representative subset of all pull requests in SWEBench) to construct TESTGENEVALLITE.

Modify SWEBench images: We start with the Docker images provided by SWEBench [2]. We first modify the images that did not build manually, by installing appropriate dependencies and

[2]https://github.com/aorwall/SWE-bench-docker

modifying requirements files so every test suite executes. Next, we install both coverage and mutation testing dependencies and modify the test commands to run with coverage instrumentation.
Extract test file pairs: After we have the execution environment for each pull request version built, we next extract code test file pairs from the code and test files run in the PR. Specifically, we extract code test file pairs by performing a heuristic match on filenames, and filtering out pull requests that do not meet our heuristic. With each file pair we perform program analysis on the test file to extract the first test, last test as context for our test completion settings. We later validate for each file pair that the gold tests cover some part of the code under test.
Filter paired tests: Next, we run code coverage for all the gold test settings (first, last, and extra) to filter out repositories where contexts were extracted incorrectly or partial test files do not run. We also filter out tests that take longer than 60 seconds to run to ensure TESTGENEVAL runs efficiently.

5.1.2 Tasks

Figure 5.5 shows an example of both testing tasks. TESTGENEVAL consists of 2 separate tasks: test generation and test completion.
Test generation: The goal of test generation is to generate an entire test suite given a file under test. We provide the necessary inputs to the model as part of the prompt. Our test generation task aligns with real-world unit testing; unit testing in practice involves writing tests for large code files in complex projects.
Test completion: The goal of test completion is to generate the next test in an existing test suite given an existing test suite and the file under test. Test completion is measured by many software testing models, and can be applied to in IDE tools, however there is no benchmark for this task. The test completion task aligns with different stages in the development life cycle; first test completion mirrors a developer starting their test suite, last test completion mirrors the finishing of their test suite and extra test completion measures whether language models can add an additional test to a test suite a developer thinks is complete. This setup is in line with Rao et al. [116], which models test generation at the method level.

5.1.3 Properties of TESTGENEVAL

We outline some of the properties that differentiate TESTGENEVAL from existing test generation benchmarks. We explain each property and motivate each property in depth.
File level test generation: Real-world unit testing involves reasoning over complex files, generating tests for a given file under test. Unlike existing benchmarks that deal with small, self-contained programs, TESTGENEVAL includes code and tests from large scale, highly starred projects. This is important as the complexity of the code under test is significantly higher than existing benchmarks, and is more representative of real-world software development. Figure 5.6 shows the lengths of code and test files on a log scale compared to CAT-LM (Chapter 3), TestEval [135], HumanEvalFix [25]. We can see that the files are much larger in TestGenEval than existing benchmarks used to evaluate LLMs, more closely resembing real-world development.
Human-written tests: Existing benchmarks such as R2E [63] measure the ability of an LLM to generate equivalence tests. While models that generate equivalence tests have the advantage

```
class Character:
  ...
  def level_up(self):
    self.level += 1

  def damage(self, health):
    self.health -= health

# TestGenEval-Full: full test suite
    generation

def test_character_setup():
  ...

def test_character_levels_up():
  ...

def test_character_damage():
  ...
```

Figure 5.3: Full test suite generation

```
class Character:
  ...
  def level_up(self):
    self.level += 1

  def damage(self, health):
    self.health -= health

# TestGenEval test prefix

def test_character_setup():
  ...

def test_character_levels_up():
  ...

# TestGenEval: test completion

def test_character_damage(self):
  ...
```

Figure 5.4: Test completion (last)

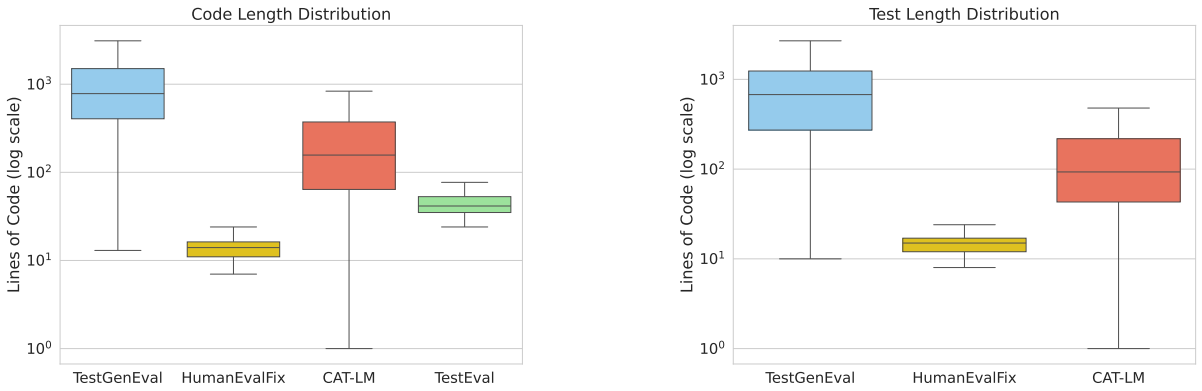
Figure 5.5: Two software testing tasks (and highlighted model generations). Full test suite generation requires knowledge of code under test setup, along with meaningful assert statements. Test completion requires understanding the code under test and current test to generate an additional test method (first, last, and extra test).

of high coverage and ease of scaling to large amounts of data, they often look different from developer-written tests [70]. Measuring the ability of LLMs to generate equivalence tests is an adjacent, but different task from measuring unit test generation capabilities. TESTGENEVAL is the first large-scale test-generation benchmark using human-written tests, which are more representative of real-world test suites.

Test suite generation: The goal of unit test generation is to generate high-quality test suites, which is correlated to high coverage and mutation score for the code under test. However, existing methods such as TestEval [135] and SWT-Bench [94] only measure the ability of an LLM to generate an individual test method rather than an entire test suite. We propose complementing individual test completion tasks with the broader test suite generation task, to better align with real-world test generation (where coverage and mutation score of the entire suite are typically measured).

Mutation score: Unlike existing benchmarks, we report mutation score in conjunction with coverage. The mutation score is empirically far more correlated with bug detection capabilities [68, 107] and much harder to hack; in order to achieve high mutation score tests must be able to discriminate non-buggy code from code with synthetic bugs introduced. This is important because even as coverage gets saturated for TESTGENEVAL, mutation score will provide a more granular measure of test quality (and thus ensure the high coverage test suites are truly high quality).

Extensibility for LLM evaluation: While traditional software engineering benchmarks such as EvoSuite [45], and Pynguin [89] can be used to evaluate test generation capabilities, these benchmarks were not built for easy extension. Unlike TESTGENEVAL, these benchmarks are not



(a) TestGenEval code lengths

(b) TestGenEval test lengths

Figure 5.6: 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.

tailored for evaluating LLMs specifically, requiring additional scaffolding to evaluate LLMs. Additionally, none of these benchmarks use Docker images for managing dependencies, resulting in a complex setup and issues with replicability (for example obtaining an old version of Java, dealing with deprecated packages). TESTGENEVAL is built off of Docker images making replicability trivial, and easily enables LLM evaluation in Python (the primary language that LLMs are trained on).

5.2 Model Performance on TESTGENEVAL

We evaluate a selection of models on TESTGENEVAL to better understand how models of different sizes and families (see Table 5.1 for list of models) perform at test generation and test completion for large scale repositories. We aim to highlight models that perform well at large scale test generation, while also understanding where the gaps are in the testing capabilities of current models, along with exploring the effects of model size on performance. We prompt each model with the maximum context window size possible, otherwise truncate the starting tokens to fit the prompt in the context window.

We report results for all models in both the full test generation (Section 5.2.3) and test completion tasks (Section 5.2.4) on TESTGENEVAL. For test suite generation we report any pass@1 (if any of the tests in the generated test suite pass), all pass@1 (if the generated test suite passes), coverage (coverage of passing tests), and mutation score (proportion of synthetic bugs introduced to code caught by test suite). Coverage and mutation score align with general test adequacy, while any pass@1 and all pass@1 provide looser metrics of utility (whether any test can be used without changing or whether the entire test suite can be used). For test completion we report pass@1 and pass@5 (whether generated test passes), along with coverage improvement from adding the generated test. These metrics similarly measure the utility of the generated test, with coverage improvement measuring if the added test tests new functionality and pass@1 and pass@5 mea-

suring if the test can be added to an existing test suite without changes.

5.2.1 Models

We evaluate a selection of models on TESTGENEVAL to better understand how models of different sizes and families perform at test generation and test completion for large scale repositories. We include a mixture of small (less than 9B params), medium (between 9B and 27B params), large (approximately 70B params) and flagship (greater than 70B params) models. This includes open source models (CodeLlama, Llama 3, DeepSeekCoder 2, Codestral, and Gemma 2) of varying sizes and GPT-4o, a state of the art closed source model.

We choose the Llama and Gemma families of models to understand the effects of size on model performance (both model families have multiple model sizes) in conjunction with their high scores on code benchmarks such as HumanEval. We also include DeepSeekCoder, CodeLlama and Codestral due to their code specialization. Finally, we include GPT-4o due to its state of the art performance on numerous code generation benchmarks.

5.2.2 Metrics

We report metrics for both our test generation (Section 5.2.2) and test completion (Section 5.2.2) tasks. We report all pass@1, any pass@1, coverage, and mutation score for test generation, and pass@k and coverage improvement for test completion. Pass@k indicates whether the generated test suite (in the case of test suite generation) or generated test can be added with no modification, while coverage and mutation score provide test adequacy metrics to measure the quality of generated tests.

Test Generation Metrics

All pass@1: All pass@1 measures if the generated test suite passes when run on the code under test. This penalizes more verbose models, as the likelihood of an error increases with each additional test generated in a test suite.
Any pass@1: Any pass@1 measures if any test in the generated test suite passes when run on the code under test. We include this to not penalize models that generate longer test suites.
Coverage: Coverage measures the proportion of lines in the file under test executed by the test suite. Ideally, a high quality test suite should execute a large percentage of the lines in the code under test.
Mutation score: The main limitation of coverage and pass@1 is that they can be potentially gamed (a model can invoke all functions in the file under test without testing anything and achieve 100% coverage and 100% pass@1). To compute mutation scores we inject synthetic bugs into the code under test. We then measure the percentage of bugs detected by the test suite (should pass on the original code and fail on the buggy code). Unlike other metrics, mutation score is much harder to game, however it is computationally costly, as we have to execute the entire test suite for each bug. We rank by coverage, as models still have relatively low coverage across the board and it is more compute efficient for the community to run (we allow for TESTGENEVAL to be run omitting mutation score).

We use cosmic-ray[3] to generate mutants and use the default set of mutation operators[4]. This default set of operators follows best practices defined by the mutation testing community [40, 100]. Cosmic ray has 565 stars and is commonly used in mutation testing research [36, 119]. We choose to not filter out mutation operators to achieve the most granular results possible (with no filtering there is only a 1.06% uncertainty in mutation score results).

Test Completion Metrics

Pass@k: Pass@k measures if any of k tests generated pass when added to the existing test suite for the code under test. We rank by this metric, as coverage improvement is near 0 for 2/3 settings for TESTGENEVAL.
Coverage improvement: Coverage improvement measures the change in line coverage when adding the generated test. Ideally, newly added tests should improve overall code coverage. We choose not to report mutation score improvement here, due to computational cost and already near 0 coverage improvement of generated tests.

5.2.3 Test generation performance of various models

Table 5.1 shows any pass@1, all pass@1 coverage and mutation score for small, medium and large models. All three smaller models perform significantly worse than their larger counterparts, with a large difference (42.6% for Llama 3.1 models and 15.6% for Gemma models in any pass@1 performance). All pass@1 penalizes how verbose a model is: DeepSeekCoder 16B on average generates a test suite with 106 lines of code and 16 methods, while Llama 3.1 70B generates an average of 324 lines of code and 36 methods. This intuitively makes sense, as the more verbose a model is, the more likely it is to generate a test with errors.

Coverage and mutation score remain low across all models. For coverage, GPT-4o performs the best, but still covers only 35.2% of the lines of code tested in TESTGENEVAL. The mutation score is even lower than the coverage, which implies that tests generated by these models cannot catch all induced bugs.

5.2.4 Test completion performance of various models

Table 5.2 shows pass@1, pass@5 and coverage improvement for first and last settings. Model performance generally increases as more of the test file is provided as context (extra pass@5 is higher than both last pass@5 and first pass@5). Similar to the full test setting, larger models tend to outperform smaller ones. An outlier is Llama 3.1 8B, with a significantly higher pass@5 in all settings than its smaller model counterparts. Codestral 22B performs the best at generating passing tests, with a pass@5 of 74.3% on the last test completion setting. Models face challenges in augmenting coverage for existing human-written test-suites, whereas they can more readily add coverage when no tests are initially present. In the final test completion setting, all models

[3] https://github.com/sixty-north/cosmic-ray
[4] https://github.com/sixty-north/cosmic-ray/tree/master/src/cosmic_ray/operators

generate virtually no new coverage, primarily testing computation paths that have already been covered.

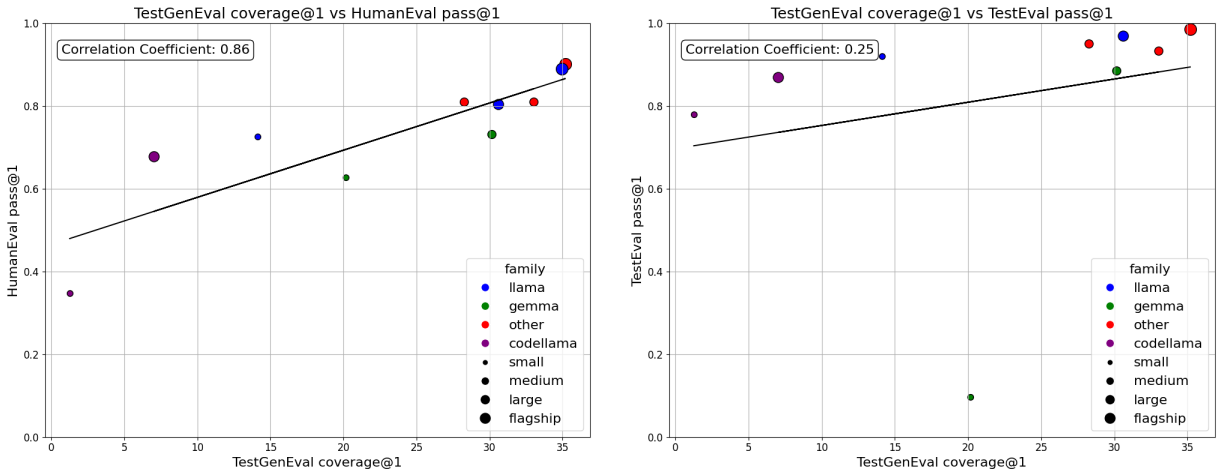
5.3 Analysis

We perform an exploratory quantitative and qualitative analysis of all results. This includes correlation with other benchmarks, effects of samples on pass@k, effects of context window size along with a qualitative analysis of differentiating problems between Codestral, GPT-4o and Llama 405B (our analysis is also in line with prior work [52]). We measure correlation with other benchmarks to understand if TESTGENEVAL adds additional information not captured by existing benchmarks, and measure the effects of sampling more and increasing context window on model performance, as both these parameters directly correlate with model cost. Finally, we examine cases where TESTGENEVAL differentiates between multiple high performing models to explore potential strengths and weaknesses of each model.

5.3.1 Quantitative Analysis

We perform an exploratory analysis of model correlation with other benchmarks (Section 5.3.1) along with the effects of sampling more (Section 5.3.1) and increasing context window (Section 5.3.1) on model performance.

Correlation with other benchmarks



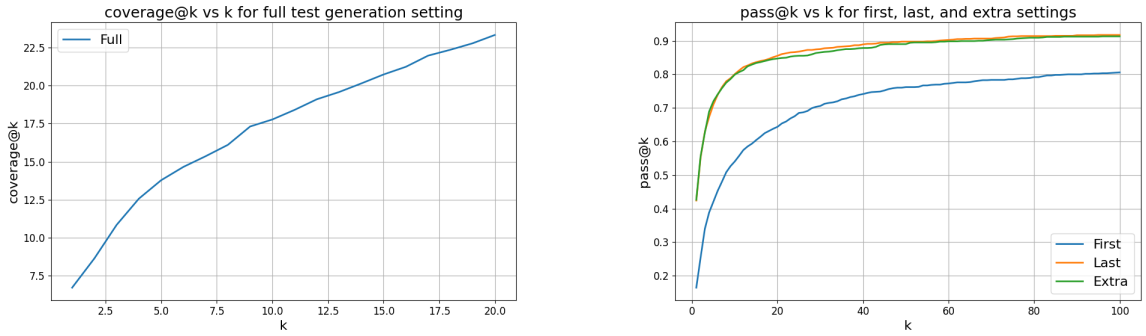
(a) Correlation with HumanEval

(b) Correlation with TestEval

Figure 5.7: Correlation with HumanEval (a SOTA code generation benchmark), and TestEval (a SOTA test generation benchmark). We find that there is a weak positive correlation between HumanEval and TESTGENEVAL, and with the exception of Gemma 9B there is a similar weak positive correlation between TestEval and TESTGENEVAL.

Figure 5.7a and Figure 5.7b display correlation between TESTGENEVAL, HumanEval (a code generation benchmark), and TestEval (a test generation benchmark). We find a weak positive correlation between TESTGENEVAL scores and other benchmarks. For HumanEval, outliers include Gemma 9B (performs better at test generation than expected given code generation performance) and CodeLlama (performs worse at test generation than expected). For TestEval, the main outlier is Gemma 9B, with reasonable test generation performance on TESTGENEVAL, but very poor performance on TestEval (it fails to properly follow the prompt format for TestEval). The positive, but non perfect correlation, between TESTGENEVAL, HumanEval, and TestEval indicates that TESTGENEVAL still adds new information that is not captured by existing benchmarks.

Effect of number of samples



(a) Coverage@k vs k (full test generation)

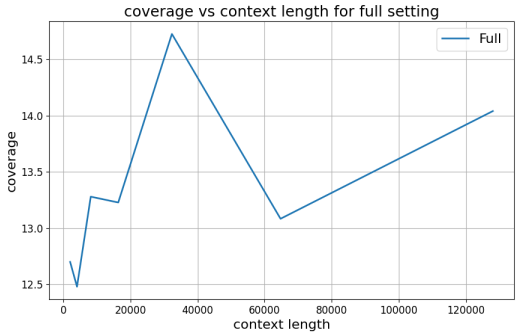
(b) Pass@k vs k (test completion)

Figure 5.8: Effect of sampling more tests for all settings on Llama 3.1 8B. Coverage@k seems to gradually increase. Pass@k also increases more for lower k values and seems to plateau after k=20.

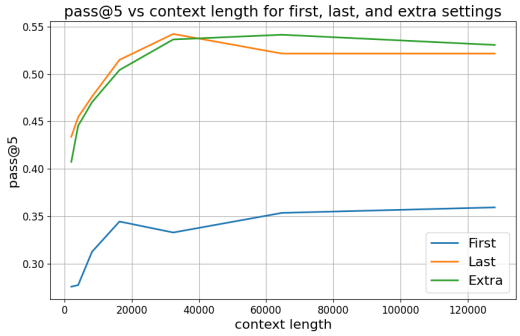
Figure 5.8a and Figure 5.8b show how performance in the full test generation and first, last and extra test completion settings changes as more tests are sampled for Llama 3.1 8B. For full test suite generation we sample 20 examples and measure coverage@k. For test completion we sample 100 examples and measure pass@k. For full test suite generation, we find that coverage seems to gradually increase, with no plateau in the first 20 generations. For test completion, we find that the first 5 samples improve performance the most, and performance gains seem to plateau after k=20 (the gain between k=20 and k=100 is minimal). This indicates that practictioners can likely improve test completion capabilities by sampling 20 examples (sampling more is not cost efficient), while for full test suite generation sampling more generally will improve performance significantly.

Effect of context window on TESTGENEVAL performance

Figure 5.9a and Figure 5.9b show the effect of context length on both our coverage (full test generation) and on pass@5 (test completion). For test generation, we find that coverage only



(a) Coverage by context length (full test generation)



(b) Pass@5 by context length (test completion)

Figure 5.9: Effect of context window for both our full setting (coverage of generated test suite) and for our test completion setting (pass@5 for our first, last and extra test completion settings). We find that context length generally helps test completion, however for test generation, even receiving parts of the file under test (measured by a lower context window) seems to be effective.

slightly improves with additional context; even seeing part of the code under test is enough for the model to generate a test suite (can mock inputs to various methods in the partial file). However for test completion context is more important, with pass@5 increasing with more context up until around 32k tokens where benefits decay. To complete tests, one must understand existing tests and their relationship with the code under test. Having the entire file helps contextualize what existing tests are testing, improving the performance of completed tests.

5.3.2 Qualitative Analysis

We outline three cases where TESTGENEVAL discriminates between GPT-4o, CodeStral and Llama 3.1 405B (only one model succeeds in each of these examples). Our goal is to better understand the strengths and weaknesses of these high performing models in an exploratory manner.

Example 1 - Test setup (only solved by GPT-4o)

```
class QuerySet(AltersData):
    def __init__(self, model=None, query=None, ...):
        ...
    def repr(self):
        return "<%s %r>" % (self.__class__.__name__, data)
    ...
```

Listing 5.1: Query set class for managing data from database.

Our first example involves a QuerySet class that manages records returned from a database. The class has no database dependencies. The initialization takes a model and query, which can

also not be set if no database is being used. This is hard to test because it involves either mocking or creating a complex model object.

```
def test_queryset_repr(self):
    queryset = QuerySet(model=None)
    assertEqual(repr(queryset), "<QuerySet []>")
```

Listing 5.2: Cut GPT-4o generated test

GPT-4o instantiates the QuerySet with no parameters (the simplest possible version of the test). It then tests all code methods, with all of the generated tests passing on the code under test.

```
from django.db import connection, models

def test_query_set_init(self):
    model = models.Model()
    qs = QuerySet(model=model)
    self.assertEqual(qs.model, model)
```

Listing 5.3: Cut Llama 3.1 405B generated test

This is not the case with both Llama 3.1 405B and Codestral 22B. Both models ignore the empty case entirely and instead hallucinate invocations or imports of models in Django (instead these models should have mocked the model object and tested the null case). All models failing to properly mock the class under test's dependencies lead to low coverage of the source file, despite GPT-4o generating passing tests all when other models fail. This indicates that existing models struggle to mock the class under test and its corresponding dependencies, a potential area for improvement for existing models.

Example 2 - Incorrectly mocking objects (only solved by Llama 3.1 405B)

```
class MigrationAutodetector:
    def __init__(self, from_state, to_state, questioner=None):
        ...
        self.existing_apps = {app for app, model in from_state.
   models}
    def changes(self, graph, trim_to_apps=None, convert_apps=None,
   migration_name=None):
        ...
```

Listing 5.4: Migration detection class to automatically detect changes needed between project states.

Our second example involves a MigrationAutodetector class that takes in two project states and measures the changes between the states passed in to automatically detect code that should be migrated.

```
def test_changes():
    class Model(models.Model):
```

```
        pass
    ...
    changes = migration.changes(graph={}, trim_to_apps=None)
    self.assertEqual(len(changes), class="syntax-number">1)
```

Listing 5.5: Cut Llama 3.1 405B generated test

Llama 3.1 405B generates passing test for the changes method, providing an empty graph and model, meaning the only difference between the from and to states is the addition of the model. The assert passes, as the method only outputs one change.

```
def test_init():
    from_state = MagicMock(spec=ProjectState)
    to_state = MagicMock(spec=ProjectState)
    autodetector = MigrationAutodetector(self.from_state, self.
   to_state, self.questioner)
    self.assertEqual(autodetector.from_state, self.from_state)
```

Listing 5.6: Cut GPT-4o generated test

GPT-4o attempts to mock the ProjectState class (passing instance checks of the ProjectState class). However, it misses the line in initialization where the model's attribute of `from_state` is accessed. This leads the setup to fail, and the MigrationAutodetector initialization fails. Codestral 22B makes a similar error, where it incorrectly mocks the ProjectState class and misses the model attribute. This indicates one potential area of improvement for high performing models is successfully mocking the class under test (important for test setup).

Example 3 - Handling class dependencies (only solved by Codestral 22B)

```
class FieldGetDbPrepValueMixin:
    ...
class Exact(FieldGetDbPrepValueMixin, BuiltinLookup):
    def process_rhs(self, compiler, connection):
        ...
```

Listing 5.7: Database classes to handle lookups and comparisons in queries.

Our final example deals with lookups and comparisons that could be applied when searching a database (for example exact match, greater than, less than, etc.). The file under test has many classes and subclasses such as `Exact`, which depends on `FieldGetDbPrepValueMixin`.

```
def test_exact_lookup():
    lhs = F('field')
    lookup = Exact(lhs, 'value')
    self.assertIn('%s', ...)
```

Listing 5.8: Cut GPT-4o generated test

GPT-4o and Llama 3.1 405B both hallucinate class invocations, rather than mocking or using the classes provided in context. F is not a correct class to pass as lhs (missing the required output field).

```
def test_exact():
    field = Field()
    lookup = Exact(field, 'value')
    self.assertEqual(lookup.lookup_name, 'exact')
```

Listing 5.9: Cut Codestral 22B generated test

Unlike GPT-4o and Llama 3.1 405B, Codestral 22B is able to understand the file under test. Codestral 22B both imports and instantiates a valid field under test that is also imported in the file under test and correctly instantiates the Exact class that has a dependency on the field. This indicates that as code bases scale, existing models struggle to generate valid class dependencies.

5.4 Limitations

We outline potential limitations with TESTGENEVAL.

Overfitting to SWEBench repositories: A potential limitation is that our benchmark is adapted from SWEBench and as a result risks overfitting to this specific dataset. Currently, this does not seem to be a major issue, as model performance is low across the board. Even once models can achieve high coverage, there is the significantly harder task of achieving high mutation score (actually catching synthetic bugs introduced into the code under test). These multiple levels of difficulty and numerous tasks help mitigate the risk of a model overfitting to any one task specifically.

Data contamination: There is also a risk of data contamination in the pretraining data of models. To further understand data contamination, we measure perplexity of 10 randomly selected tests in TESTGENEVAL for Llama 3.1 8B and common frequent, and non recent code from GitHub with similar lengths. We find that the perplexity of this common GitHub code is lower than the 10 tests from TESTGENEVAL (1.6 vs 2.0), indicating that data is unlikely to be contaminated. This is further supported by the low performance of all models on TESTGENEVAL across the board.

Compute cost of mutation score: One other limitation is the compute cost of computing mutation score. Each synthetic bug we introduce to the code under test, requires an additional test suite execution. However, our results show that coverage and mutation score are highly correlated. Setting a timeout of one hour per mutation testing run, we only timeout on 20% of files, and get an average uncertainty of 1.1%. We provide an option to run TESTGENEVAL without mutation, enabling those who lack compute to still benefit from TESTGENEVAL.

5.5 Conclusion

This chapter supports a key contribution of my thesis: improving the evaluation of unit test generation techniques through more realistic benchmarks and new metrics such as mutation

score which more closely align with test quality. We introduce TESTGENEVAL, the first file-level benchmark for test generation and test completion. We release a lite version consisting of 160 code-test file pairs and a full benchmark comprising 1,210 file pairs from real open-source projects. Considering real-world settings, we employ coverage and mutation score to evaluate the models in our benchmark, as these metrics are closely related to the real-world quality of test suites. We perform a comprehensive evaluation of both open and closed source models on TESTGENEVAL to better understand how well existing models perform on test suite generation and test completion. We find that existing models struggle to generate high quality test suites and add to existing test suites (pass@5 below 60% for test completion and coverage below 40% for test suite generation). Additionally, we perform an exploratory qualitative analysis of high performing models, finding that models struggle with test setup and mocking. Overall, we believe TESTGENEVAL provides a complementary dataset to existing test generation datasets, offering a more challenging and larger-scale version of current benchmarks. This benchmark is used in the next chapter to evaluate the performance of my test generation agent on large scale programs (Chapter 6).

Table 5.1: Full test suite generation. All results shown for temperature=0.2. Larger models generally perform better at test suite generation, however all models struggle to achieve high coverage and mutation score.

Model	All Pass@1	Any Pass@1	Coverage	Mutation
Small Models				
CodeLlama 7B	3.2%	4.1%	1.2%	0.5%
Gemma 9B	3.5%	42.1%	20.2%	9.0%
Llama 3.1 8B	3.1%	30.5%	14.1%	6.8%
Medium Models				
DeepSeekCoder 16B	23.0%	63.7%	28.2%	12.1%
Gemma 27B	7.4%	57.7%	30.1%	14.6%
Codestral 22B	**26.8%**	72.7%	33.0%	14.2%
Large Models				
CodeLlama 70B	14.0%	22.7%	7.0%	2.5%
Llama 3.1 70B	7.8%	60.7%	30.6%	15.4%
Flagship Models				
GPT-4o	7.5%	64.0%	**35.2%**	**18.8%**
Llama 3.1 405B	17.7%	**73.1%**	35.0%	16.4%

Table 5.2: Pass rates for different models in first, and last settings. Pass@1 is for temperature=0.2, Pass@5 is for temperature=0.8. Codestral 22B outperforms all models across pass@1 and pass@5 (other than first test completion pass@5), solving between 60-70% of all tasks.

	First			Last		
Model	**Pass@1**	**Pass@5**	**+Cov**	**Pass@1**	**Pass@5**	**+Cov**
Small Models						
CodeLlama 7B	4.2%	19.8%	6.7%	6.9%	24.0%	0.0%
Gemma 9B	8.4%	18.0%	6.7%	21.4%	46.4%	0.1%
Llama 3.1 8B	14.4%	33.3%	12.8%	32.0%	54.3%	0.1%
Medium Models						
DeepSeekCoder 16B	18.6%	47.0%	19.0%	17.0%	62.8%	0.2%
Gemma 27B	12.7%	33.7%	13.6%	32.2%	62.2%	0.1%
Codestral 22B	**38.3%**	61.7%	24.0%	**50.4%**	**74.3%**	0.4%
Large Models						
CodeLlama 70B	0.5%	30.2%	11.2%	0.9%	50.7%	0.0%
Llama 3.1 70B	19.3%	46.4%	18.7%	35.0%	61.9%	**0.5%**
Flagship Models						
GPT-4o	31.9%	**63.5%**	**26.9%**	32.6%	66.6%	**0.5%**
Llama 3.1 405B	32.1%	57.7%	21.6%	42.6%	72.3%	0.3%

6 Feedback Driven, Agentic Test Suite Generation

In this chapter, my collaborators and I leverage both test execution and code coverage feedback to build a unit test generation agent.[1] Our approach is evaluated on TestGenEval, thus more closely representing real-world development at scale. Unlike prior approaches, we designed TestForge with cost in mind (costing $0.63 per file, cheaper than other agentic approaches), processing execution feedback at the file level rather than the method level, allowing the agent to plan out and execute multiple edits or generate multiple test cases. This work highlights the importance of combining execution feedback with the relation between code and test files, while also evaluating on a large-scale benchmark.

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. Unlike pipeline approaches, agents can freely choose which files they view, and how they interact with the environment (whether to run the tests or not, get coverage information, etc.). Such a system can autonomously explore code, edit tests, and self-reflect on its outputs. Recent techniques like CoverUp [111] and HITS [138] 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 [62].

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. We specifically target regression testing and assume the code under test is correct (measuring bug finding capabilities is difficult in practice due to the oracle problem in automated test generation techniques). 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 to generate a large test suite. 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

[1] Work submitted to ICSE 2026 [60]

coverage as input to the 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 my large-scale benchmark TestGenEval [62] (see Chapter 5 for more details). For search-based techniques, we compare against Pynguin [89] (pure genetic programming) and CodaMosa [79] (genetic programming augmented with LLMs). We also compare against non-agentic LLM-based baselines CAT-LM [116], and GPT-4o [102]. 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 percentage points 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 likely 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[2] 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.

The contributions of this chapter are:

- TestForge, a state-of-the-art and cost-effective test generation agentic system. TestForge uses dynamic feedback and an interactive 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]https://github.com/All-Hands-AI/OpenHands/tree/main/evaluation/benchmarks/testgeneval

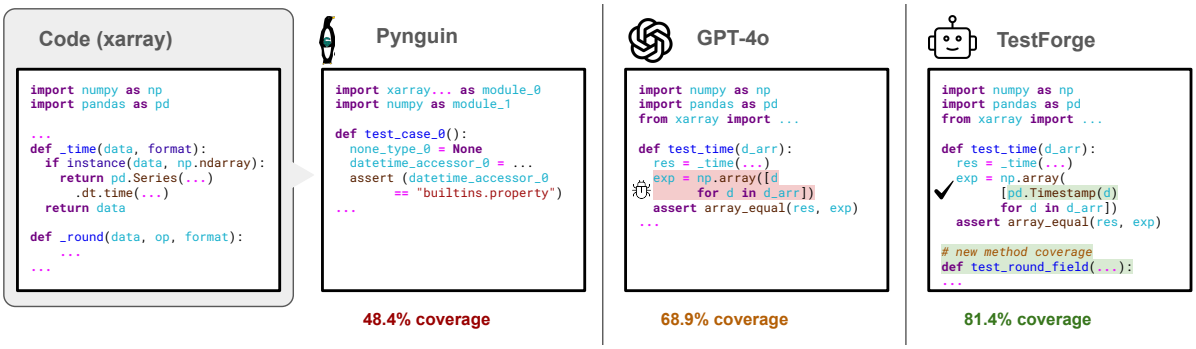


Figure 6.1: Example of tests generated by different approaches for timing and rounding functionality in pydata/xarray. GPT-4o generates a failing test, which TestForge fixes, while also adding additional coverage improving tests.

6.1 Illustrative Example

We begin by motivating and illustrating our approach with an example. The left-hand-side of Figure 6.1 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 the 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 6.1 shows a test for this file produced by Pynguin [89], 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 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. Part of the challenge is that projects in TestGenEval are much more complex, making program analysis more flaky and less reliable (evidenced by the segmentation faults that Pynguin runs into on some of the TestGenEval programs).

Simply and directly zero-shot prompting GPT-4o [102] 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 a mutation score of 47.4% on the file, outperforming Pynguin; the names and structure 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

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

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 (which is one of the three pieces of execution feedback used in TestForge).

Other LLM agent approaches such as HITS [138] and CoverUp [111] use dependency analysis and program slicing as part of their agentic loop (the core loop where the LLM decides the next step to take). 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 observations 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 6.1 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.

6.2 TestForge

In this section, we provide an overview of TestForge, including the technical design details and the general agentic workflow. Section 6.2.1 shows the design of TestForge and an illustrative example of the agentic workflow. Section 6.2.2 provides more technical details behind each tool call our agent can make, Section 6.2.3 provides an overview of all execution feedback we can get from the environment, and Section 6.2.4 provides details about our implementation of TestForge and integration with OpenHands.

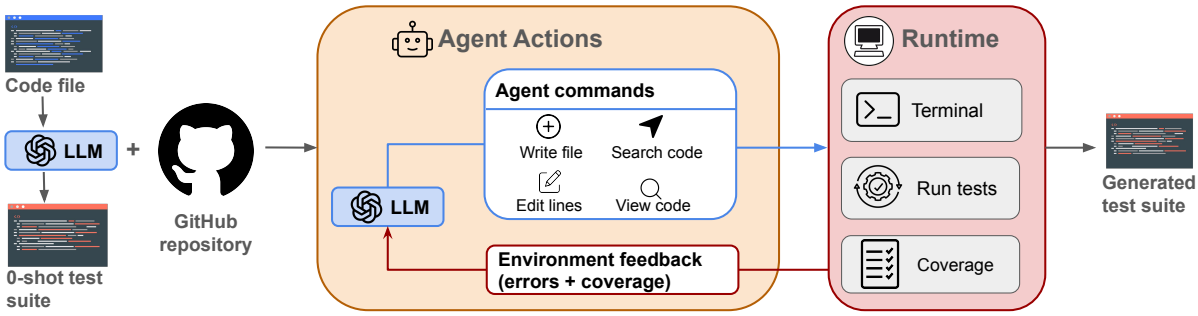


Figure 6.2: Overview of TestForge. We start by generating a zero-shot test suite and allowing our agent to interact with the repository with the generated test suite. We include the ability to search code, view code, write and edit files along with environment capabilities to run commands and tests. The output is a full test suite for the code file under test.

6.2.1 Overview

Figure 6.2 shows a full overview of TestForge. The input of TestForge is a code file to be tested (dependencies can be retrieved as part of the agentic loop); 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 6.4.3). Following this, TestForge enters the agentic loop. TestForge can either perform an *agent command* (actions listed in the box, labeled "Agent Actions") or *interact with the environment* (actions listed in the 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 (see Section 6.2.2 for the full list of actions) and their corresponding input and output formats as tools when calling OpenAI's API. Our actions are consistent with CodeAct [136] (a state-of-the-art code generation agent), 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 (any tool call that executes a command or generated tests), 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 [140], 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 6.1: TestForge prompt template

Listing 6.1 shows an abridged version of our agent prompt (this is different from the self-reflection that occurs after an environment tool call is executed). 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; generating more than 20 overwhelms model context, while fewer than 20 tests generally corresponds with low 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 6.2: TestForge bash tool call JSON definition

```
{
    "id": "call_h3lCE3GtZaKOke9hqPCdv0Wy",
    "type": "function",
    "function": {
```

```
        "name": "execute_bash",
        "arguments": "{"command": "..."}"
    }
}
```

Listing 6.3: TestForge bash tool call LLM output

Listing 6.2 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. There is also associated Python code that implements the logic for executing a terminal command. Listing 6.3 shows an example of TestForge of calling the `execute_bash` tool defined in Listing 6.2; the LLM supplies the required arguments and the name of the tool.

6.2.2 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 [136], which achieves state-of-the-art performance on SWEBench [65].

Search code

An important function for code agents is searching *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.

Write file

We provide TestForge the ability to create any file or overwrite any existing file. This is also in line with prior work [136, 144]. In practice, TestForge primarily writes Python test files. There is no limit on how long the generated file can be. This enables TestForge to overwrite the entire test suite when the zero-shot test suite has many errors.

Edit lines

We provide editing functionality to TestForge for any file (also in line with prior work [136, 144]). 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

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

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.

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 [144] and CodeAct [136].

6.2.3 Environment Feedback

In addition to agent actions, the system also provides feedback from the environment to TestForge. We define environment feedback in LiteLLM in the same way as agent actions (the difference being that environment feedback involves executing commands or generated tests). Here, we outline sources of important feedback for TestForge when generating tests.

Test execution feedback

One important source of execution feedback is the output from test execution. We provide the agent with the command to execute the generated test suite. 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.

Code coverage feedback

In addition to test execution feedback, we also provide coverage report feedback. We provide the agent with the command to create a 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.

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).

6.2.4 Implementation

TestForge is implemented in approximately 4.3k LOC of Python; our replication package is available at: `https://github.com/All-Hands-AI/OpenHands/tree/main/evaluation/benchmarks/testgeneval`.

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 [62] 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 [136]. 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.

6.3 Experimental Setup

We compare TestForge with Pynguin [89] and CodaMosa [79], current state-of-the-art unit test generation tools and GPT-4o 0-shot prompting on the TestGenEval benchmark [62]. We ask the following research questions:

RQ1: Runtime Performance: How well does TestForge perform at generating high coverage regression 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 (to understand what are the most frequently used actions by TestForge) and a small case study of tests generated by TestForge and each of the baselines (to understand the strengths and weaknesses of each approach in regards to readability and maintainability).

6.3.1 Dataset

We evaluated all baselines and TestForge on TestGenEval (see Chapter 5 for more details). We choose TestGenEval over existing benchmarks such as CAT-LM [116], TestEval [135] and Hu-

manEvalFix [25], 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) [121, 135]. TestGenEval provides a benchmark that existing models perform poorly on, where we can measure the impact of execution feedback.

6.3.2 Baselines

We compare TestForge against multiple classical and LLM baselines. We run TestForge for 25 iterations (the average time per data point of TestForge is 447 seconds), and classical baselines for 600 seconds (a common cutoff used in prior work [79, 111]). For classical baselines, we choose Pynguin [89] and CodaMosa [79]. 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 [79] 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 [88] to GPT-4o [103] (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 [102] and CAT-LM [116]. 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 [62], 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 [34], HITS [138] and CoverUp [111], 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

[5] https://github.com/se2p/pynguin/issues/81

our price range.

6.3.3 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.

Runtime Metrics

We use the same set of runtime metrics introduced with TestGenEval (see Chapter 5 for more description on each 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 [69, 101]. High mutation score indicates a test suite is capable of catching future 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.

Lexical Metrics

We also supplement TestGenEval with lexical metrics to measure the similarity between generated test suites and developer-written test suites. We use the same set of lexical metrics as CAT-LM (see Chapter 3 for more details on each metric).

CodeBLEU [117] 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 [82] 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 [96, 116] as a

[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 6.1: TestGenEval runtime results on the full 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.

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.

6.4 Results and Analysis

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

6.4.1 RQ1: Runtime Performance

Table 6.1 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

Model	CodeBLEU	ROUGE	Edit Similarity
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**

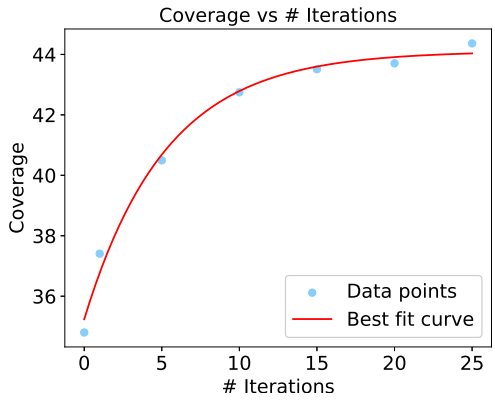
Table 6.2: TestGenEval lexical results for all 1210 programs in the dataset. All LLM-based approaches achieve similar scores on all lexical metrics (besides the low ROUGE score of CAT-LM), but genetic programming approaches achieve far lower lexical performance.

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).

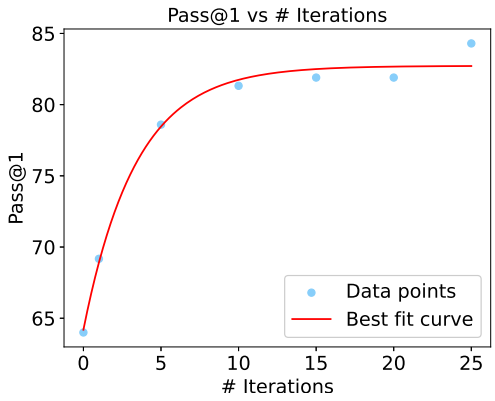
We also report runtime metrics in comparison against CodaMosa and Pynguin on the 45 data point subset that uses Python 3.10 (a requirement for both CodaMosa and Pynguin). For cases where Pynguin and CodaMosa do not generate tests, we mark both the coverage and mutation score of the respective approach 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 [79]). 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 than both genetic programming approaches by greater than 10 percentage points and 6 percentage points when compared to 0-shot. The difference for mutation score is even greater with a greater than 25 percentage point 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 difference of 20.9% and 12.2% respectively and a mutation score difference of 23.2 percentage points and 30.8 percentage points respectively. All



(a) Coverage with TestForge iterations



(b) Pass@1 with TestForge iterations

Figure 6.3: 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.

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 reasonably efficient. We evaluate the impact of running fewer iterations (providing potential cost savings) in Section 6.4.3.

6.4.2 RQ2: Lexical Performance

We report lexical performance as a measure of how similar generated test suites are to developer written test suites. Table 6.2 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 (evidenced by their much higher lexical scores), especially when compared to search-based test generation approaches. LLMs are pretrained on human code, whereas search-based approaches have no notion of "naturalness".

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 percentage points. 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. Higher lexical performance indicates that LLM based approaches generate test suites more similar to developer written test suites than search based approaches.

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

Table 6.3: 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.

6.4.3 RQ3: Design Decisions

We discuss the impact of various design decisions involved in TestForge. Specifically, we measure both coverage and pass@1 varying number of iterations and measure the impact of including the zero-shot test file as the starting point for TestForge.

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 6.3 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. Pass@1 increases in fewer iterations than coverage because the criteria is less fine-grained (if only one generated test passes in the test suite, pass@1 is 1).

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 6.3 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.

6.4.4 RQ4: Behavior

We perform both a quantitative analysis of actions taken by TestForge (Section 6.4.4) to understand why TestForge performs the way it does and a qualitative analysis of tests generated by

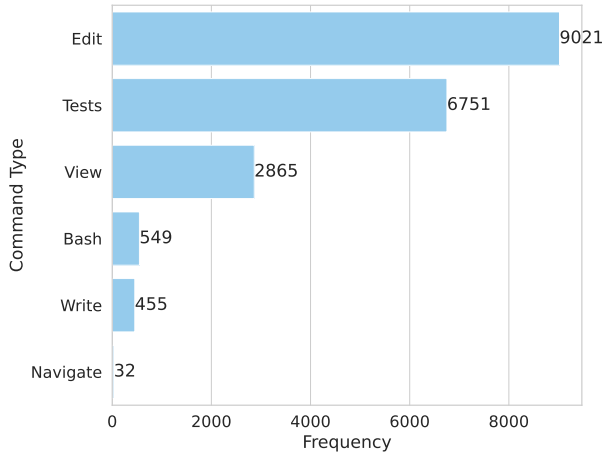


Figure 6.4: Frequency of TestForge commands taken while generating tests for all 1210 programs in TestGenEval. The most common actions are editing and executing the generated test suite, indicating an iterative approach to test suite refinement.

TestForge and other baselines (Section 6.4.4) to understand the strengths and weaknesses of each approach in regards to readability and maintainability of generated tests.

Quantitative Action Analysis

Figure 6.4 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 6.1). Navigation is much less frequently used, suggesting that the complex dependency analysis used by prior work [111, 138] 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.

Qualitative Readability Analysis

Following prior work [116], 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.

Pynguin: Pynguin generates a test that covers the method under test. However, the test contains

```
@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 6.4: 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 6.5: 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 6.6: 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 6.7: TestForge (repaired) test

Figure 6.5: 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.

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 Pynguin'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.

Overall, we find that search based approaches struggle with both readability and maintainability of their generated test suites. Tests generated by Pynguin and CodaMosa suffer from poor variable names, poor test names, and tests that are marked as expected to fail. Meanwhile, LLM based approaches generate tests with high quality variable and test names, with failing tests being filtered out rather than marked as expected to fail.

6.5 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 [115].

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 of 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.

6.6 Conclusion

In this chapter, I leverage test execution feedback in combination with the coupling between code and test files to design a unit test generation agent that scales to large scale repositories. Unlike prior work, my approach is cost-effective ($0.63 per file), due to taking feedback into account in parallel and allowing the agent to look through dependencies it sees as important. The evaluation is also much more realistic, with TestGenEval being sourced from 11 large scale open source projects.

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]Replication: https://github.com/All-Hands-AI/OpenHands/tree/main/evaluation/benchmarks/testgeneval

7 Conclusions and Final Remarks

Software testing is fundamentally different from code generation; test code has a different structure than traditional source code and a strong coupling with the code under test. Thus, existing approaches applying language models (both in pretraining and fine-tuning) to software testing can be improved with additional domain-specific, non-local context: the source file, test prefix and execution data from running the generated test. We show the importance of this context in two separate tasks: unit test generation and mutation testing. For unit test generation, we pretrain a model with the relationship between source code and test code as a first class citizen, finding that a model pretrained this way outperforms existing models with orders of magnitude more parameters and training budgets. For mutation testing, we add additional test method context to existing predictive mutation testing approaches, finding that this context enables meaningfully higher performance. I also bootstrap LLMs with execution information, including the output from running tests and coverage information to build an agent for unit test generation that scales to large scale projects. In addition to this, I also developed a large-scale benchmark (Chapter 5) that better measures the performance of LLMs on test generation and test completion tasks in a real-world setting. This benchmark reveals flaws in existing approaches, which do not scale in cost to this larger setting. We help overcome these issues with TestForge, the first software testing agent that successfully leverages execution feedback in a cost-effective manner.

My work on pretraining LLMs for unit test generation (Chapter 3), fine-tuning for mutation testing (Chapter 4) and the development of TestForge (Chapter 6) have implications for the broader software engineering community. The dataset and pretraining objective of CAT-LM can be easily incorporated into existing LLM pretraining pipelines, with improvements even when scaled up to larger models (approximately 70B parameters). This dataset of code and corresponding tests is relatively high quality, sourced from highly starred Java and Python projects, with a heuristic to align code and test files. To our knowledge, there was no dataset similar to this for the community to use. Our predictive mutation testing approach was the first to be evaluated in a way that is practically meaningful to developers. We compute end user time even confirming all surviving mutants, showing that this approach saves over 66% time even with this conservative evaluation. Interestingly, this approach to mutation testing requires a smaller language model; LLM inference is too time costly to be practical (running test suite only takes 300ms, so model inference time needs to be low to be practical). My software testing agent is the first that scales to large scale testing datasets in a cost-efficient manner. It is also integrated into the highly popular OpenHands framework to enable easy extension and evaluation by the software engineering community.

There are still many open challenges in software testing that are yet to be solved. While my

agentic approach to software testing was able to get a moderately high coverage of 44.4% and mutation score of 33.8%, there is still room for improvement. Often TestForge would get stuck in a loop trying to repair errors in one or a few tests, preventing it from progressing and increasing coverage. Reasoning over long files was still lacking too, often TestForge would focus only on the first part of the file rather than the entire file. Ideally, high quality automated test generation should mimic the coverage and completeness of large open source repositories; the repositories we used in TESTGENEVAL had a baseline coverage of greater than 80%. More research on long context code reasoning could help here, for example teaching models to summarize functionality of large codebases or perform cross file code edits. Benchmarks like SWEBench [65] are a starting point, but more complex benchmarks dealing with large codebases and multi-file edits could help advance the state-of-the-art of LLMs.

Another analogous direction would be to develop larger code and test datasets that resemble real-world tests. One idea is to perform synthetic data generation, whereby we can prompt a model for a problem, code solution and tests and then perform rejection sampling to filter out bad examples. This could help solve the issue of scale, with there only being a limited number of code and tests available on open source. Scaling synthetic data will hopefully improve downstream performance on real-world tasks, if the synthetic data is representative of real-world code and tests. Along a similar line, one can prompt and perform rejection sampling on repositories without tests or to add additional tests for repositories with tests. These can also be used as part of supervised fine-tuning for test generation.

Additionally, one benefit of software testing is that there are many automated metrics that can be used to evaluate test adequacy (pass@1, coverage, mutation score). One promising direction is to use reinforcement learning with these execution metrics, to better optimize test generation capabilities. A combination of coverage, tests passing and mutation score could be used in combination with a benchmark like TestGenEval to better align current models with generating high quality tests. RLEF [48] performed a similar approach for code contests where tests passing and code compilation were used as reward signals to better align Llama [42] models with high quality solutions.

Improving test generation and code generation can be complementary as well. A high quality test generator can be combined with a code generation technique to self-improve both. Similar to the idea of GANs, one can have the test generator try to generate high coverage and mutation score tests that break the code solution and the code generator can refine its solutions based on the output of the test generators. This verification scheme can be used to improve the quality of code solutions, potentially even using tests as a contract to provide some guarantees of correctness of the code solution. This can also complement other test generation solutions at inference time like CodeT [24], which uses consensus sets to cluster various code solutions with consistent test cases (larger sets are more likely to be correct).

One other major challenge that all test generation approaches suffer from the oracle problem. Both classical and neural test generation approaches assume the code under test is correct, which is often not true. One promising direction here is to leverage alternative sources of information, for example documentation or code comments that more closely align with the developer specification. Supplementing test generation models with this specification might help break the assumption that the code under test is correct. Another promising direction is using differential testing, where we have two implementations of the same piece of software. These implementa-

tions can be used to create a differential "oracle", in other words tests that pass on one implementation and fail on the other are likely to reveal a bug in one of the implementations. While this is a limitation of all automated test generation approaches, these approaches can still be used in the context of regression testing, where we can use generated tests to catch future regressions. LLMs based approaches, like the ones outlined in my dissertation, also help overcome the readability issue of classical test generation approaches, with generated tests being more similar to human written tests (as evidenced by qualitative studies and lexical metrics).

Another challenge is evaluating readability and maintainability of code. While LLMs are trained on developer code and have a strong notion of naturalness, it is still not fully clear how readable or maintainable LLM generated code is. There have been some human studies on this [32, 124], but these studies are relatively small scale or outdated for current LLMs. More research needs to be done into understanding how LLM generated code affects maintainability of code in the long run, and to understand the limitations of existing LLM based approaches such as the ones outlined in this thesis. While this is a limitation of all LLM automated test generation work, in general LLM generated code is still more readable than search based techniques; LLMs are by construction pretrained to produce code that is as statistically similar to human written code as possible.

The thesis contributes in the following ways:

1. It presents a new method for pretraining models for test generation, that considers the relationship between source code and test code.
2. It provides an approach to automatically classify mutants as detected or undetected without executing the test suite by leveraging additional *test* context.
3. It evaluates all provided techniques with metrics and experiments that are practically meaningful developers, not considered in prior work.
4. It introduces a benchmark for evaluating test generation approaches that is sourced from large scale open source repositories and thus more closely resembles real-world test generation.
5. It demonstrates the effectiveness of adding execution context to test generation models, which enables us to generate high quality test suites for large scale projects.

In summary, my work on software testing has shown that LLMs can be effectively used for software testing tasks, with the right context and feedback. This work has implications for the broader software engineering community, with a new benchmark, pretraining objective, fine-tuning technique and agent that can be used to improve software testing tasks. There are still many open challenges in software testing that can be addressed with future work, including scaling to larger codebases, improving test generation and code generation and combining the two to self-improve both techniques.

Bibliography

[1] GitHub REST API. URL https://docs.github.com/en/rest. 3.3.1

[2] GPT-neox Toolkit. URL https://github.com/EleutherAI/gpt-neox. 3.1, 3.4.2

[3] SentencePiece. URL https://github.com/google/sentencepiece. 3.1, 3.4.1

[4] TheFuzz: Fuzzy String Matching in Python. URL https://github.com/seatgeek/thefuzz. 3.3.2

[5] GitHub Copilot, 2021. URL https://github.com/features/copilot. 1, 3

[6] ChatGPT, November 2022. URL https://openai.com/blog/chatgpt/. 1

[7] Alireza Aghamohammadi and Seyed-Hassan Mirian-Hosseinabadi. The threat to the validity of predictive mutation testing: The impact of uncovered mutants. *CoRR*, abs/2005.11532, 2020. doi: 10.48550/arXiv.2005.11532. 2.2.2

[8] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified Pre-training for Program Understanding and Generation. In *North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, NAACL-HLT '21, pages 2655–2668, Jun 2021. doi: 10.18653/v1/2021.naacl-main.211. 4.1.2

[9] Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. In *Automated Software Engineering*, ASE '23, 2023. doi: 10.1145/3551349.3559555. 4.4.2

[10] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Onward!*, pages 143–153. ACM, 2019. 3.3.1

[11] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta, 2024. URL https://arxiv.org/abs/2402.09171. 2.1.1

[12] M. Aniche, C. Treude, and A. Zaidman. How developers engineer test cases: An observational study. *IEEE Transactions on Software Engineering*, 48(12):4925–4946, dec 2022. ISSN 1939-3520. doi: 10.1109/TSE.2021.3129889. 1, 3

[13] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *arXiv:abs/2108.07732*, 2021. 2.2.3

[14] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: Fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019. 2.1.1

[15] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Survey*, 51 (3):50–88, 2018. 1

[16] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient Training of Language Models to Fill in the Middle. *CoRR*, abs/2207.14255, 2022. 1, 3.5.1

[17] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 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*, ESEC/FSE '15, page 179–190, 2015. 1, 2.1.1, 2.2.1

[18] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *International Conference on Software Engineering*, ICSE '15, page 559–562, 2015. 1, 2.1.1, 2.2.1

[19] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry - A study at facebook. In *International Conference on Software Engineering: Software Engineering in Practice*, ICSE '18, pages 268–277. IEEE, 2021. doi: 10.1109/ICSE-SEIP52600.2021.00036. 1

[20] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. Unit test generation using generative ai : A comparative performance analysis of autogeneration tools, 2024. URL `https://arxiv.org/abs/2312.10622`. 2.2.3, 5

[21] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. *SIGSOFT Software Engineering Notes*, 27(4):123–133, 2002. 2.2.1

[22] Carolin Brandt and Andy Zaidman. Developer-centric test amplification: The interplay between automatic generation human exploration. *Empirical Software Engineering*, 27 (4), 2022. ISSN 1382-3256. 1, 2.2.1

[23] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are Few-Shot learners. In *Advances in Neural Information Processing Systems*, pages 1877–1901, 2020. 2.1.3

[24] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and

Weizhu Chen. CodeT: Code generation with generated tests. In *ICLR*, 2023. 7

[25] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *CoRR*, abs/2107.03374, 2021. 1, 1.2, 2.2.3, 3.5.1, 5, 5.1.3, 6.3.1

[26] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation, 2024. URL `https://arxiv.org/abs/2305.04764`. 2.2.1

[27] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *International Conference on Software Engineering*, ICSE '19, pages 736–747, 2019. 2.2.1

[28] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014. URL `https://arxiv.org/abs/1412.3555`. 2.1.3

[29] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming*, ICFP '00, page 268–279, 2000. 2.2.1

[30] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: A practical mutation testing tool for Java (demo). In *International Symposium on Software Testing and Analysis*, ISSTA '16, pages 449–452, 2016. doi: 10.1145/2931037.2948707. 2.1.2

[31] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211, 2014. doi: 10.1109/ISSRE.2014.11. 2.1.1

[32] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 107–118, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786838. URL `https://doi.org/10.1145/2786805.2786838`. 7

[33] Ermira Daka, José Miguel Rojas, and Gordon Fraser. 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, pages 57–67, 2017. 1.2, 2.2.1

[34] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and

Michel C. Desmarais. Effective test generation using pre-trained large language models and mutation testing, 2023. URL https://arxiv.org/abs/2308.16557. 2.2.1, 6.3.2

[35] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022. 3.1, 3.6

[36] Sourav Deb, Kush Jain, Rijnard van Tonder, Claire Le Goues, and Alex Groce. Syntax is all you need: A universal-language approach to mutant generation. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. doi: 10.1145/3643756. URL https://doi.org/10.1145/3643756. 5.2.2

[37] Sourav Deb, Kush Jain, Rijnard van Tonder, Claire Le Goues, and Alex Groce. Syntax is all you need: A universal-language approach to mutant generation. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. doi: 10.1145/3643756. URL https://doi.org/10.1145/3643756. 2.2.2

[38] DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence, 2024. URL https://arxiv.org/abs/2406.11931. 2.1.3

[39] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, Apr 1978. doi: 10.1109/C-M.1978.218136. 2.1.2, 4.5

[40] Anna Derezinska and Konrad Halas. Experimental evaluation of mutation testing approaches to python programs. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 156–164, 2014. doi: 10.1109/ICSTW.2014.24. 5.2.2

[41] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu Lahiri. Toga: A neural method for test oracle generation. In *International Conference on Software Engineering*, ICSE '22, page 2130–2141, 2022. 1, 2.2.1, 2.2.1, 2.2.3, 3, 3.5.1, 3.5.2, 5

[42] Abhimanyu Dubey et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783. 7

[43] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP*, EMNLP '20, pages 1536–1547, November 2020. doi: 10.18653/v1/2020.findings-emnlp.139. 4.1.1, 4.1.2

[44] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *Conference on Offensive Technologies*, WOOT

'20, pages 10–10, 2020. 2.2.1

[45] Gordon Fraser and Andrea Arcuri. 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*, ESEC/FSE '11, page 416–419, 2011. 1, 2.1.1, 2.2.1, 5, 5.1.3

[46] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013. doi: 10.1109/TSE.2012.14. 1

[47] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A Generative Model for Code Infilling and Synthesis. *CoRR*, abs/2204.05999, 2022. 1

[48] Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning, 2025. URL `https://arxiv.org/abs/2410.02089`. 7

[49] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. How hard does mutation analysis have to be, anyway? In *Software Reliability Engineering*, pages 216–227, 2015. doi: 10.1109/ISSRE.2015.7381815. 4.5

[50] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, Regular-Expression-based tool for multi-language mutant generation. In *International Conference on Software Engineering*, ICSE '18, pages 25–28, 2018. doi: 10.1145/3183440.3183485. 2.1.2

[51] Alex Groce, Kush Jain, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. Looking for Lacunae in Bitcoin Core's Fuzzing Efforts. In *International Conference on Software Engineering: Software Engineering in Practice*, ICSE '22, 2022. doi: 10.1145/3510457.3513072. 2.2.2

[52] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024. 2.2.3, 5.3

[53] R.G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, 1977. doi: 10.1109/TSE.1977.231145. 4.5

[54] Vincent J Hellendoorn and Premkumar Devanbu. Are Deep Neural Networks the best choice for modeling source code? In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '17, pages 763–773, 2017. doi: 10.1145/3106237.3106290. 4.4.2

[55] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *NeurIPS Datasets and Benchmarks*, 2021. 2.2.3

[56] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847, 2012. doi: 10.1109/ICSE.2012.6227135. 1

[57] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735. 2.1.3

[58] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *CoRR*, arXiv:2203.15556, 2022. 3.1, 3.4.2

[59] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with Hybrid lexical and syntactical information. *Empirical Software Engineering*, 25(3):2179–2217, 2020. 3.5.1

[60] Kush Jain and Claire Le Goues. Testforge: Feedback-driven, agentic test suite generation, 2025. URL https://arxiv.org/abs/2503.14713. 1.3, 1

[61] Kush Jain, Uri Alon, Alex Groce, and Claire Le Goues. Contextual predictive mutation testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 250–261, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703270. doi: 10.1145/3611643.3616289. URL https://doi.org/10.1145/3611643.3616289. 1.3, 1, 4.2, 3, 4

[62] Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark, 2024. URL https://arxiv.org/abs/2410.00752. 1, 1.2, 1.3, 1, 6, 6.2.4, 6.3, 6.3.2

[63] Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2e: Turning any github repository into a programming agent environment. In *ICML*, 2024. 2.2.3, 5, 5.1.3

[64] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2010. doi: 10.1109/TSE.2010.62. 4.5

[65] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66. 2.2.3, 5, 6.2.2, 7

[66] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013. doi: 10.1109/ICSE.2013.6606613. 1.2

[67] René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *International Symposium on Software Testing and Analysis*, ISSTA '14, pages 433–436. Association for Computing Machinery, 2014. doi: 10.1145/2610384.2628053. 2.1.2, 4.2.2

[68] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gor-

don Fraser. Are mutants a valid substitute for real faults in software testing? In *Symposium on Foundations of Software Engineering*, FSE '14, pages 654–665, 2014. URL https://doi.org/10.1145/2635868.2635929. 1, 2.1.2, 2.2.2, 2.2.3, 4.4.1, 5, 5.1.3

[69] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 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*, FSE 2014, page 654–665, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635929. URL https://doi.org/10.1145/2635868.2635929. 6.3.3

[70] Rajeswari Hita Kambhamettu, John Billos, Tomi Oluwaseun-Apo, Benjamin Gafford, Rohan Padhye, and Vincent J. Hellendoorn. On the naturalness of fuzzer-generated code. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 506–510, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3527972. URL https://doi.org/10.1145/3524842.3527972. 5.1.3

[71] Samuel J. Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. Prioritizing mutants to guide mutation testing. In *International Conference on Software Engineering*, ICSE '22, 2022. doi: 10.1145/3510003.3510187. 2.2.2, 4.1

[72] Ameya Ketkar, Daniel Ramos, Lazaro Clapp, Raj Barik, and Murali Krishna Ramanathan. A lightweight polyglot code transformation language. *Proc. ACM Program. Lang.*, 8 (PLDI), June 2024. doi: 10.1145/3656429. URL https://doi.org/10.1145/3656429. 2.1.2

[73] V. Khorikov. *Unit Testing Principles, Practices, and Patterns: Effective testing styles, patterns, and reliable automation for unit testing, mocking, and integration testing with examples in C#*. Manning, 2020. ISBN 9781617296277. URL https://books.google.com/books?id=CbvZyAEACAAJ. 2.1.1

[74] Jinhan Kim, Juyoung Jeon, Shin Hong, and Shin Yoo. Predictive mutation analysis via the natural language channel in source code. *ACM Trans. Softw. Eng. Methodol.*, 31(4), jul 2022. ISSN 1049-331X. doi: 10.1145/3510417. URL https://doi.org/10.1145/3510417. 1, 1.2, 2.2.2, 4, 4.1.1, 4.2, 4.2.1, 4.2.2, 4.2.3, 4.2.4

[75] Pavneet Singh Kochhar, Tegawendé F. Bissyandé, David Lo, and Lingxiao Jiang. An empirical study of adoption of software testing in open source projects. In *International Conference on Quality Software*, ICQS '13, pages 103–112, 2013. 3.3.2

[76] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 560–564, 2015. doi: 10.1109/SANER.2015.7081877. 1.2

[77] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Annual Meeting of the Association for Computational Linguistics*, ACL '18, pages 66–75, 2018. 3.4.1

[78] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A Neural model for generating natural language summaries of program subroutines. In *International Conference on Software Engineering*, ICSE '19, pages 795–806, 2019. 3.5.1

[79] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha 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)*, pages 919–931, 2023. doi: 10.1109/ICSE48619.2023.00085. 6, 6.3, 6.3.2, 6.4.1

[80] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! 2023. 3, 3.5.1

[81] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 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*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3608128. URL https://doi.org/10.1145/3597503.3608128. 1

[82] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Conference on Text Summarization Branches Out*, pages 74–81, 2004. 1.2, 2.2.3, 3.5.1, 6.3.3

[83] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. Exploring and evaluating hallucinations in llm-powered code generation, 2024. 1

[84] Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems, 2023. URL https://arxiv.org/abs/2306.03091. 5

[85] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv:abs/1907.11692*, 2019. 2.1.3

[86] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: a map of code duplicates on GitHub. In *Proceedings of the ACM on Programming Languages*, volume 1 of *OOPSLA '17*, pages 1–28, 2017. 3.3.1

[87] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun

Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021. 3.5.2

[88] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *NeurIPS Datasets and Benchmarks*, 2021. 6.3.2

[89] Stephan Lukasczyk and Gordon Fraser. Pynguin: automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, page 168–172, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392235. doi: 10.1145/3510454.3516829. URL https://doi.org/10.1145/3510454.3516829. 1, 2.1.1, 5, 5.1.3, 6, 6.1, 6.3, 6.3.2

[90] Wei Ma, Mengjie Zhao, Ezekiel Soremekun, Qiang Hu, Jie M. Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. GraphCode2Vec: Generic code embedding via lexical and program dependence analyses. In *Mining Software Repositories*, MSR '22, pages 524–536, 2022. doi: 10.1145/3524842.3528456. 4.1.1

[91] David MacIver, Zac Hatfield-Dodds, and Many Contributors. Hypothesis: A New Approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019. 2.2.1

[92] Dongyu Mao, Lingchao Chen, and Lingming Zhang. An extensive study on Cross-Project predictive mutation testing. In *Software Testing, Validation and Verification*, ICST '19, pages 160–171, 2019. doi: 10.1109/ICST.2019.00025. 2.2.2

[93] Tukaram B Muske, Ankit Baid, and Tushar Sanas. Review efforts reduction by partitioning of static analysis warnings. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 106–115, 2013. doi: 10.1109/SCAM.2013.6648191. 1.2

[94] Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. Code agents are state of the art software testers, 2024. URL https://arxiv.org/abs/2406.12952. 2.2.3, 5.1.3

[95] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 1–5, 2022. doi: 10.1145/3524842.3528470. 1

[96] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Learning deep semantics for test completion. In *International Conference on Software Engineering*, ICSE '23, page 2111–2123, 2023. 1, 1.2, 2.2.1, 2.2.3, 3, 3.5.1, 3.5.1, 3.5.2, 3.7.2, 5, 6.3.3

[97] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A Conversational Paradigm for Program Synthesis. *CoRR*,

abs/2203.13474, 2022. 1, 3, 3.4.1, 3.4.2, 3.5.1

[98] A. Jefferson Offutt and Roland H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation Testing for the New Century*, pages 34–44. Springer, 2001. doi: 10.5555/571305.571314. 2.2.2

[99] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, 1996. doi: 10.1145/227607.227610. 2.2.2, 4.5

[100] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, apr 1996. ISSN 1049-331X. doi: 10.1145/227607.227610. URL https://doi.org/10.1145/227607.227610. 5.2.2

[101] A Jeeerson Ooutt and Jeerey M Voas. Subsumption of condition coverage techniques by mutation testing. 1996. URL https://api.semanticscholar.org/CorpusID:9492620. 6.3.3

[102] OpenAI. Gpt-4 technical report, 2023. 2.1.3, 6, 6.1, 6.3.2

[103] OpenAI et al. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774. 2.1.3, 6.3.2

[104] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for Java. In *Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, 2007. 2.2.1

[105] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities. In *International Conference on Software Maintenance and Evolution*, pages 523–533, 2020. 1

[106] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 537–548, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180183. URL https://doi.org/10.1145/3180155.3180183. 2.2.2, 2.2.3

[107] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *International Conference on Software Engineering*, ICSE '18, pages 537–548, 2018. doi: 10.1145/3180155.3180183. 1.2, 2.1.2, 4.4.1, 5, 5.1.3

[108] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *ACL*, pages 311–318. ACL, 2002. 2.2.3

[109] Goran Petrovic and Marko Ivankovic. State of mutation testing at Google. In *International Conference on Software Engineering: Software Engineering in Practice*, ICSE '18, pages 163–171, 2018. doi: 10.1145/3183519.3183521. 2.1.2, 4.1, 4.5

[110] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *Software Testing, Verification and Validation Workshops*, ICSTW '18, pages 47–53, 2018. doi: 10.1109/ICSTW.2018.00027. 2.1.2

[111] Juan Altmayer Pizzorno and Emery D. Berger. Coverup: Coverage-guided llm-based test generation, 2025. URL https://arxiv.org/abs/2403.16218. 2.2.1, 6, 6.1, 6.3.2, 6.4.4

[112] Martin Popel and Ondrej Bojar. Training Tips for the Transformer Model. *CoRR*, abs/1804.00247, 2018. doi: 10.48550/arXiv.1804.00247. 4.2.3

[113] Habibur Rahman and Saqib Ameen. How is testing related to single statement bugs?, 2024. URL https://arxiv.org/abs/2403.18226. 2.1.1

[114] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. Melt: Mining effective lightweight transformations from pull requests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1516–1528. IEEE, 2023. 2.1.2

[115] Daniel Ramos, Claudia Mamede, Kush Jain, Paulo Canelas, Catarina Gamboa, and Claire Le Goues. Are large language models memorizing bug benchmarks?, 2024. URL https://arxiv.org/abs/2411.13323. 6.5

[116] Nikitha Rao, Kush Jain, U. Alon, Claire Le Goues, and Vincent Joshua Hellendoorn. Cat-lm training language models on aligned code and tests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 409–420, Los Alamitos, CA, USA, sep 2023. IEEE Computer Society. doi: 10.1109/ASE56229.2023.00193. URL https://doi.ieeecomputersociety.org/10.1109/ASE56229.2023.00193. 1, 1.2, 1.3, 1, 3.5.1, 3.5.1, 3.7, 4.5, 5, 5.1.2, 6, 6.3.1, 6.3.2, 6.3.3, 6.4.4

[117] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020. URL https://arxiv.org/abs/2009.10297. 1.2, 2.1.3, 3.5.1, 6.3.3

[118] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. 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, pages 23–32, 2011. 2.2.1

[119] Ana B. Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. Mutation testing in the wild: findings from github. *Empirical Softw. Engg.*, 27(6), November 2022. ISSN 1382-3256. doi: 10.1007/s10664-022-10177-8. URL https://doi.org/10.1007/s10664-022-10177-8. 5.2.2

[120] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Association for Computational Linguistics*, ACL '16, pages 1715–1725, 2016. doi: 10.18653/v1/P16-1162. 4.2.3

[121] Yuling Shi, Songsong Wang, Chengcheng Wan, and Xiaodong Gu. From code to correctness: Closing the last mile of code generation with hierarchical debugging, 2024. URL https://arxiv.org/abs/2410.01215. 6.3.1

[122] Hugo Henrique Fumero de Souza, Igor Wiese, Igor Steinmacher, and Reginaldo Ré. A characterization study of testing contributors and their contributions in open source projects. In *Brazilian Symposium on Software Engineering*, SBES '22, pages 95–105, 2022. 3.3.2

[123] Alexey Svyatkovskiy, Sarah Fakhoury, Negar Ghorbani, Todd Mytkowicz, Elizabeth Dinella, Christian Bird, Jinu Jang, Neel Sundaresan, and Shuvendu K. Lahiri. Program Merge Conflict Resolution via Neural Transformers. In *Symposium on the Foundations of Software Engineering*, FSE '22, pages 822–833, 2022. doi: 10.1145/3540250.3549163. 4.1.1, 4.1.2

[124] Wannita Takerngsaksiri, Micheal Fu, Chakkrit Tantithamthavorn, Jirat Pasuksmit, Kun Chen, and Ming Wu. Code readability in the age of large language models: An industrial case study from atlassian, 2025. URL https://arxiv.org/abs/2501.11264. 7

[125] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Slav Petrov, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul R. Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, Alexandre Frechette, Charlotte Smith, Laura Culp, Lev Proleev, Yi Luan, Xi Chen, James Lottes, Nathan Schucher, Federico Lebron, Alban Rrustemi, Natalie Clay, Phil Crone, Tomas Kocisky, Jeffrey Zhao, Bartek Perz, Dian Yu, Heidi Howard, Adam Bloniarz, Jack W. Rae, Han Lu, Laurent Sifre, Marcello Maggioni, Fred Alcober, Dan Garrette, Megan Barnes, Shantanu Thakoor, Jacob Austin, Gabriel Barth-Maron, William Wong, Rishabh Joshi, Rahma Chaabouni, Deeni Fatiha, Arun Ahuja, Ruibo Liu, Yunxuan Li, Sarah Cogan, Jeremy Chen, Chao Jia, Chenjie Gu, Qiao Zhang, Jordan Grimstad, Ale Jakse Hartman, Martin Chadwick, Gaurav Singh Tomar, Xavier Garcia, Evan Senter, Emanuel Taropa, Thanumalayan Sankaranarayana Pillai, Jacob Devlin, Michael Laskin, Diego de Las Casas, Dasha Valter, Connie Tao, Lorenzo Blanco, Adrià Puigdomènech Badia, David Reitter, Mianna Chen, Jenny Brennan, Clara Rivera, Sergey Brin, Shariq Iqbal, Gabriela Surita, Jane Labanowski, Abhi Rao, Stephanie Winkler, Emilio Parisotto, Yiming Gu, Kate Olszewska, Yujing Zhang, Ravi Addanki, Antoine Miech, Annie Louis, Laurent El Shafey, Denis Teplyashin, Geoff Brown, Elliot Catt, Nithya Attaluri, Jan Balaguer, Jackie Xiang, Pidong Wang, Zoe Ashwood, Anton Briukhov, Albert Webson, Sanjay Ganapathy, Smit Sanghavi, Ajay Kannan, Ming-Wei Chang, Axel Stjerngren, Josip Djolonga, Yuting Sun, Ankur Bapna, Matthew Aitchison, Pedram Pejman, Henryk Michalewski, Tianhe Yu, Cindy Wang, Juliette Love, Junwhan

Ahn, Dawn Bloxwich, Kehang Han, Peter Humphreys, Thibault Sellam, James Bradbury, Varun Godbole, Sina Samangooei, Bogdan Damoc, Alex Kaskasoli, Sébastien M. R. Arnold, Vijay Vasudevan, Shubham Agrawal, Jason Riesa, Dmitry Lepikhin, Richard Tanburn, Srivatsan Srinivasan, Hyeontaek Lim, Sarah Hodkinson, Pranav Shyam, Johan Ferret, Steven Hand, Ankush Garg, Tom Le Paine, Jian Li, Yujia Li, Minh Giang, Alexander Neitz, Zaheer Abbas, Sarah York, Machel Reid, Elizabeth Cole, Aakanksha Chowdhery, Dipanjan Das, Dominika Rogozińska, Vitaly Nikolaev, Pablo Sprechmann, Zachary Nado, Lukas Zilka, Flavien Prost, Luheng He, Marianne Monteiro, Gaurav Mishra, Chris Welty, Josh Newlan, Dawei Jia, Miltiadis Allamanis, Clara Huiyi Hu, Raoul de Liedekerke, Justin Gilmer, Carl Saroufim, Shruti Rijhwani, Shaobo Hou, Disha Shrivastava, Anirudh Baddepudi, Alex Goldin, Adnan Ozturel, Albin Cassirer, Yunhan Xu, Daniel Sohn, Devendra Sachan, Reinald Kim Amplayo, Craig Swanson, Dessie Petrova, Shashi Narayan, Arthur Guez, Siddhartha Brahma, Jessica Landon, Miteyan Patel, Ruizhe Zhao, Kevin Villela, Luyu Wang, Wenhao Jia, Matthew Rahtz, Mai Giménez, Legg Yeung, Hanzhao Lin, James Keeling, Petko Georgiev, Diana Mincu, Boxi Wu, Salem Haykal, Rachel Saputro, Kiran Vodrahalli, James Qin, Zeynep Cankara, Abhanshu Sharma, Nick Fernando, Will Hawkins, Behnam Neyshabur, Solomon Kim, Adrian Hutter, Priyanka Agrawal, Alex Castro-Ros, George van den Driessche, Tao Wang, Fan Yang, Shuo yiin Chang, Paul Komarek, Ross McIlroy, Mario Lučić, Guodong Zhang, Wael Farhan, Michael Sharman, Paul Natsev, Paul Michel, Yong Cheng, Yamini Bansal, Siyuan Qiao, Kris Cao, Siamak Shakeri, Christina Butterfield, Justin Chung, Paul Kishan Rubenstein, Shivani Agrawal, Arthur Mensch, Kedar Soparkar, Karel Lenc, Timothy Chung, Aedan Pope, Loren Maggiore, Jackie Kay, Priya Jhakra, Shibo Wang, Joshua Maynez, Mary Phuong, Taylor Tobin, Andrea Tacchetti, Maja Trebacz, Kevin Robinson, Yash Katariya, Sebastian Riedel, Paige Bailey, Kefan Xiao, Nimesh Ghelani, Lora Aroyo, Ambrose Slone, Neil Houlsby, Xuehan Xiong, Zhen Yang, Elena Gribovskaya, Jonas Adler, Mateo Wirth, Lisa Lee, Music Li, Thais Kagohara, Jay Pavagadhi, Sophie Bridgers, Anna Bortsova, Sanjay Ghemawat, Zafarali Ahmed, Tianqi Liu, Richard Powell, Vijay Bolina, Mariko Iinuma, Polina Zablotskaia, James Besley, Da-Woon Chung, Timothy Dozat, Ramona Comanescu, Xiance Si, Jeremy Greer, Guolong Su, Martin Polacek, Raphaël Lopez Kaufman, Simon Tokumine, Hexiang Hu, Elena Buchatskaya, Yingjie Miao, Mohamed Elhawaty, Aditya Siddhant, Nenad Tomasev, Jinwei Xing, Christina Greer, Helen Miller, Shereen Ashraf, Aurko Roy, Zizhao Zhang, Ada Ma, Angelos Filos, Milos Besta, Rory Blevins, Ted Klimenko, Chih-Kuan Yeh, Soravit Changpinyo, Jiaqi Mu, Oscar Chang, Mantas Pajarskas, Carrie Muir, Vered Cohen, Charline Le Lan, Krishna Haridasan, Amit Marathe, Steven Hansen, Sholto Douglas, Rajkumar Samuel, Mingqiu Wang, Sophia Austin, Chang Lan, Jiepu Jiang, Justin Chiu, Jaime Alonso Lorenzo, Lars Lowe Sjösund, Sébastien Cevey, Zach Gleicher, Thi Avrahami, Anudhyan Boral, Hansa Srinivasan, Vittorio Selo, Rhys May, Konstantinos Aisopos, Léonard Hussenot, Livio Baldini Soares, Kate Baumli, Michael B. Chang, Adrià Recasens, Ben Caine, Alexander Pritzel, Filip Pavetic, Fabio Pardo, Anita Gergely, Justin Frye, Vinay Ramasesh, Dan Horgan, Kartikeya Badola, Nora Kassner, Subhrajit Roy, Ethan Dyer, Víctor Campos, Alex Tomala, Yunhao Tang, Dalia El Badawy, Elspeth White, Basil Mustafa, Oran Lang, Abhishek Jindal, Sharad Vikram, Zhitao Gong, Sergi Caelles, Ross Hemsley, Gregory Thornton, Fangxiaoyu Feng, Wojciech Stokowiec, Ce Zheng,

Phoebe Thacker, Çağlar Ünlü, Zhishuai Zhang, Mohammad Saleh, James Svensson, Max Bileschi, Piyush Patil, Ankesh Anand, Roman Ring, Katerina Tsihlas, Arpi Vezer, Marco Selvi, Toby Shevlane, Mikel Rodriguez, Tom Kwiatkowski, Samira Daruki, Keran Rong, Allan Dafoe, Nicholas FitzGerald, Keren Gu-Lemberg, Mina Khan, Lisa Anne Hendricks, Marie Pellat, Vladimir Feinberg, James Cobon-Kerr, Tara Sainath, Maribeth Rauh, Sayed Hadi Hashemi, Richard Ives, Yana Hasson, YaGuang Li, Eric Noland, Yuan Cao, Nathan Byrd, Le Hou, Qingze Wang, Thibault Sottiaux, Michela Paganini, Jean-Baptiste Lespiau, Alexandre Moufarek, Samer Hassan, Kaushik Shivakumar, Joost van Amersfoort, Amol Mandhane, Pratik Joshi, Anirudh Goyal, Matthew Tung, Andrew Brock, Hannah Sheahan, Vedant Misra, Cheng Li, Nemanja Rakićević, Mostafa Dehghani, Fangyu Liu, Sid Mittal, Junhyuk Oh, Seb Noury, Eren Sezener, Fantine Huot, Matthew Lamm, Nicola De Cao, Charlie Chen, Gamaleldin Elsayed, Ed Chi, Mahdis Mahdieh, Ian Tenney, Nan Hua, Ivan Petrychenko, Patrick Kane, Dylan Scandinaro, Rishub Jain, Jonathan Uesato, Romina Datta, Adam Sadovsky, Oskar Bunyan, Dominik Rabiej, Shimu Wu, John Zhang, Gautam Vasudevan, Edouard Leurent, Mahmoud Alnahlawi, Ionut Georgescu, Nan Wei, Ivy Zheng, Betty Chan, Pam G Rabinovitch, Piotr Stanczyk, Ye Zhang, David Steiner, Subhajit Naskar, Michael Azzam, Matthew Johnson, Adam Paszke, Chung-Cheng Chiu, Jaume Sanchez Elias, Afroz Mohiuddin, Faizan Muhammad, Jin Miao, Andrew Lee, Nino Vieillard, Sahitya Potluri, Jane Park, Elnaz Davoodi, Jiageng Zhang, Jeff Stanway, Drew Garmon, Abhijit Karmarkar, Zhe Dong, Jong Lee, Aviral Kumar, Luowei Zhou, Jonathan Evens, William Isaac, Zhe Chen, Johnson Jia, Anselm Levskaya, Zhenkai Zhu, Chris Gorgolewski, Peter Grabowski, Yu Mao, Alberto Magni, Kaisheng Yao, Javier Snaider, Norman Casagrande, Paul Suganthan, Evan Palmer, Geoffrey Irving, Edward Loper, Manaal Faruqui, Isha Arkatkar, Nanxin Chen, Izhak Shafran, Michael Fink, Alfonso Castaño, Irene Giannoumis, Wooyeol Kim, Mikołaj Rybiński, Ashwin Sreevatsa, Jennifer Prendki, David Soergel, Adrian Goedeckemeyer, Willi Gierke, Mohsen Jafari, Meenu Gaba, Jeremy Wiesner, Diana Gage Wright, Yawen Wei, Harsha Vashisht, Yana Kulizhskaya, Jay Hoover, Maigo Le, Lu Li, Chimezie Iwuanyanwu, Lu Liu, Kevin Ramirez, Andrey Khorlin, Albert Cui, Tian LIN, Marin Georgiev, Marcus Wu, Ricardo Aguilar, Keith Pallo, Abhishek Chakladar, Alena Repina, Xihui Wu, Tom van der Weide, Priya Ponnapalli, Caroline Kaplan, Jiri Simsa, Shuangfeng Li, Olivier Dousse, Fan Yang, Jeff Piper, Nathan Ie, Minnie Lui, Rama Pasumarthi, Nathan Lintz, Anitha Vijayakumar, Lam Nguyen Thiet, Daniel Andor, Pedro Valenzuela, Cosmin Paduraru, Daiyi Peng, Katherine Lee, Shuyuan Zhang, Somer Greene, Duc Dung Nguyen, Paula Kurylowicz, Sarmishta Velury, Sebastian Krause, Cassidy Hardin, Lucas Dixon, Lili Janzer, Kiam Choo, Ziqiang Feng, Biao Zhang, Achintya Singhal, Tejasi Latkar, Mingyang Zhang, Quoc Le, Elena Allica Abellan, Dayou Du, Dan McKinnon, Natasha Antropova, Tolga Bolukbasi, Orgad Keller, David Reid, Daniel Finchelstein, Maria Abi Raad, Remi Crocker, Peter Hawkins, Robert Dadashi, Colin Gaffney, Sid Lall, Ken Franko, Egor Filonov, Anna Bulanova, Rémi Leblond, Vikas Yadav, Shirley Chung, Harry Askham, Luis C. Cobo, Kelvin Xu, Felix Fischer, Jun Xu, Christina Sorokin, Chris Alberti, Chu-Cheng Lin, Colin Evans, Hao Zhou, Alek Dimitriev, Hannah Forbes, Dylan Banarse, Zora Tung, Jeremiah Liu, Mark Omernick, Colton Bishop, Chintu Kumar, Rachel Sterneck, Ryan Foley, Rohan Jain, Swaroop Mishra, Jiawei Xia, Taylor Bos, Geoffrey Cideron, Ehsan

Amid, Francesco Piccinno, Xingyu Wang, Praseem Banzal, Petru Gurita, Hila Noga, Premal Shah, Daniel J. Mankowitz, Alex Polozov, Nate Kushman, Victoria Krakovna, Sasha Brown, MohammadHossein Bateni, Dennis Duan, Vlad Firoiu, Meghana Thotakuri, Tom Natan, Anhad Mohananey, Matthieu Geist, Sidharth Mudgal, Sertan Girgin, Hui Li, Jiayu Ye, Ofir Roval, Reiko Tojo, Michael Kwong, James Lee-Thorp, Christopher Yew, Quan Yuan, Sumit Bagri, Danila Sinopalnikov, Sabela Ramos, John Mellor, Abhishek Sharma, Aliaksei Severyn, Jonathan Lai, Kathy Wu, Heng-Tze Cheng, David Miller, Nicolas Sonnerat, Denis Vnukov, Rory Greig, Jennifer Beattie, Emily Caveness, Libin Bai, Julian Eisenschlos, Alex Korchemniy, Tomy Tsai, Mimi Jasarevic, Weize Kong, Phuong Dao, Zeyu Zheng, Frederick Liu, Fan Yang, Rui Zhu, Mark Geller, Tian Huey Teh, Jason Sanmiya, Evgeny Gladchenko, Nejc Trdin, Andrei Sozanschi, Daniel Toyama, Evan Rosen, Sasan Tavakkol, Linting Xue, Chen Elkind, Oliver Woodman, John Carpenter, George Papamakarios, Rupert Kemp, Sushant Kafle, Tanya Grunina, Rishika Sinha, Alice Talbert, Abhimanyu Goyal, Diane Wu, Denese Owusu-Afriyie, Cosmo Du, Chloe Thornton, Jordi Pont-Tuset, Pradyumna Narayana, Jing Li, Sabaer Fatehi, John Wieting, Omar Ajmeri, Benigno Uria, Tao Zhu, Yeongil Ko, Laura Knight, Amélie Héliou, Ning Niu, Shane Gu, Chenxi Pang, Dustin Tran, Yeqing Li, Nir Levine, Ariel Stolovich, Norbert Kalb, Rebeca Santamaria-Fernandez, Sonam Goenka, Wenny Yustalim, Robin Strudel, Ali Elqursh, Balaji Lakshminarayanan, Charlie Deck, Shyam Upadhyay, Hyo Lee, Mike Dusenberry, Zonglin Li, Xuezhi Wang, Kyle Levin, Raphael Hoffmann, Dan Holtmann-Rice, Olivier Bachem, Summer Yue, Sho Arora, Eric Malmi, Daniil Mirylenka, Qijun Tan, Christy Koh, Soheil Hassas Yeganeh, Siim Põder, Steven Zheng, Francesco Pongetti, Mukarram Tariq, Yanhua Sun, Lucian Ionita, Mojtaba Seyedhosseini, Pouya Tafti, Ragha Kotikalapudi, Zhiyu Liu, Anmol Gulati, Jasmine Liu, Xinyu Ye, Bart Chrzaszcz, Lily Wang, Nikhil Sethi, Tianrun Li, Ben Brown, Shreya Singh, Wei Fan, Aaron Parisi, Joe Stanton, Chenkai Kuang, Vinod Koverkathu, Christopher A. Choquette-Choo, Yunjie Li, TJ Lu, Abe Ittycheriah, Prakash Shroff, Pei Sun, Mani Varadarajan, Sanaz Bahargam, Rob Willoughby, David Gaddy, Ishita Dasgupta, Guillaume Desjardins, Marco Cornero, Brona Robenek, Bhavishya Mittal, Ben Albrecht, Ashish Shenoy, Fedor Moiseev, Henrik Jacobsson, Alireza Ghaffarkhah, Morgane Rivière, Alanna Walton, Clément Crepy, Alicia Parrish, Yuan Liu, Zongwei Zhou, Clement Farabet, Carey Radebaugh, Praveen Srinivasan, Claudia van der Salm, Andreas Fidjeland, Salvatore Scellato, Eri Latorre-Chimoto, Hanna Klimczak-Plucińska, David Bridson, Dario de Cesare, Tom Hudson, Piermaria Mendolicchio, Lexi Walker, Alex Morris, Ivo Penchev, Matthew Mauger, Alexey Guseynov, Alison Reid, Seth Odoom, Lucia Loher, Victor Cotruta, Madhavi Yenugula, Dominik Grewe, Anastasia Petrushkina, Tom Duerig, Antonio Sanchez, Steve Yadlowsky, Amy Shen, Amir Globerson, Adam Kurzrok, Lynette Webb, Sahil Dua, Dong Li, Preethi Lahoti, Surya Bhupatiraju, Dan Hurt, Haroon Qureshi, Ananth Agarwal, Tomer Shani, Matan Eyal, Anuj Khare, Shreyas Rammohan Belle, Lei Wang, Chetan Tekur, Mihir Sanjay Kale, Jinliang Wei, Ruoxin Sang, Brennan Saeta, Tyler Liechty, Yi Sun, Yao Zhao, Stephan Lee, Pandu Nayak, Doug Fritz, Manish Reddy Vuyyuru, John Aslanides, Nidhi Vyas, Martin Wicke, Xiao Ma, Taylan Bilal, Evgenii Eltyshev, Daniel Balle, Nina Martin, Hardie Cate, James Manyika, Keyvan Amiri, Yelin Kim, Xi Xiong, Kai Kang, Florian Luisier, Nilesh Tripuraneni, David Madras, Mandy Guo, Austin Waters, Oliver Wang, Joshua Ainslie, Jason

Baldridge, Han Zhang, Garima Pruthi, Jakob Bauer, Feng Yang, Riham Mansour, Jason Gelman, Yang Xu, George Polovets, Ji Liu, Honglong Cai, Warren Chen, XiangHai Sheng, Emily Xue, Sherjil Ozair, Adams Yu, Christof Angermueller, Xiaowei Li, Weiren Wang, Julia Wiesinger, Emmanouil Koukoumidis, Yuan Tian, Anand Iyer, Madhu Gurumurthy, Mark Goldenson, Parashar Shah, MK Blake, Hongkun Yu, Anthony Urbanowicz, Jennimaria Palomaki, Chrisantha Fernando, Kevin Brooks, Ken Durden, Harsh Mehta, Nikola Momchev, Elahe Rahimtoroghi, Maria Georgaki, Amit Raul, Sebastian Ruder, Morgan Redshaw, Jinhyuk Lee, Komal Jalan, Dinghua Li, Ginger Perng, Blake Hechtman, Parker Schuh, Milad Nasr, Mia Chen, Kieran Milan, Vladimir Mikulik, Trevor Strohman, Juliana Franco, Tim Green, Demis Hassabis, Koray Kavukcuoglu, Jeffrey Dean, and Oriol Vinyals. Gemini: A family of highly capable multimodal models, 2023. 3

[126] Nikolai Tillmann and Peli de Halleux. Pex - white box test generation for .net. In *Tests and Proofs*, volume 4966 of *TAP '08*, pages 134–153, April 2008. 2.2.1

[127] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models. *CoRR*, abs/2302.13971, 2023. 3.4.2

[128] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers. *CoRR*, abs/2009.05617, 2020. 1, 1.2, 2.2.1, 5

[129] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating accurate assert statements for unit test cases using pretrained transformers. *CoRR*, abs/2009.05634, 2020. URL `https://arxiv.org/abs/2009.05634`. 1.2

[130] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using Mutant schemata. *ACM SIGSOFT Software Engineering Notes*, 18(3):139–148, 1993. doi: 10.1145/154183.154265. 2.2.2

[131] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017. doi: 10.5555/3295222.3295349. 2.1.3, 4, 4.1.2

[132] Victor Veloso and Andre Hora. Characterizing high-quality test methods: a first empirical study. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 265–269, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3529092. URL `https://doi.org/10.1145/3524842.3529092`. 1

[133] Johannes Villmow, Jonas Depoix, and Adrian Ulges. ConTest: A Unit Test Completion Benchmark featuring Context. In *Workshop on Natural Language Processing for Programming*, pages 17–25, August 2021. 1, 2.2.1, 2.2.3

[134] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. *CoRR*, abs/2002.08653, 2020. doi: 10.48550/arXiv.2002.08653. 4.1.1

[135] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. Testeval: Benchmarking large language models for test case generation, 2024. URL https://arxiv.org/abs/2406.04531. 1, 1.2, 2.2.3, 5, 5.1.3, 6.3.1

[136] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024. URL https://arxiv.org/abs/2402.01030. 6.2.1, 6.2.2, 6.2.2, 6.2.2, 6.2.2, 6.2.4

[137] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021. 3.5.1, 3.5.2

[138] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. Hits: High-coverage llm-based unit test generation via method slicing, 2024. URL https://arxiv.org/abs/2408.11324. 2.2.1, 6, 6.1, 6.3.2, 6.4.4

[139] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. *CoRR*, abs/2002.05800, 2020. 1, 2.2.1, 2.2.3, 3.5.1

[140] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903. 6.2.1

[141] Leandro von Werra. Codeparrot. https://github.com/huggingface/transformers/tree/main/examples/research_projects/codeparrot. 3.1, 3.3.2

[142] Robert White and Jens Krinke. Reassert: Deep learning for assert generation. *CoRR*, abs/2011.09784, 2020. 2.2.1

[143] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. A Systematic Evaluation of Large Language Models of Code. *CoRR*, abs/2202.13169, 2022. 3.4.1, 3.4.2

[144] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793. 6.2.2, 6.2.2, 6.2.2

[145] Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, ICML'20, 2020. doi: 10.5555/3524938.3525939. 4.1.1

[146] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. Predictive mutation testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 342–353, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2931038. URL https://doi.org/10.1145/2931037.2931038. 1, 1.2, 2.2.2, 4, 4.1.1, 4.2, 4.2.1

[147] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions, 2024. URL https://arxiv.org/abs/2406.15877. 5