Thesis Proposal: Exploiting Test Structure to Enhance Language Models for Software Testing

Kush Jain

April 17, 2024

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

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

Abstract

Software testing is an integral part of software development. However, testing faces challenges due to the time-consuming nature of writing high-quality tests, leading to poorly maintained test suites and lower overall software quality. This sparked research in automated test generation, including classical and neural-based methods. Classical tools, 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.

I focus on how we can incorporate domain-specific properties such as the strong coupling between source and test files along with important test execution data to improve the 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. My thesis statement is: We can exploit the structure of test code and close relationship between code and test files to enable the practical application of language models to software testing in both pretraining and fine-tuning. This approach can be used to a) generate useful unit test cases b) identify weaknesses in existing test suites and c) improve test suites to overcome found weaknesses.

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 demonstrates the effectiveness of adding execution context to test generation models, which enables us to generate mutant killing tests.
4. It evaluates all provided techniques with metrics and experiments that are practically meaningful developers, not considered in prior work

Work I have already completed (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 incorporating both source and test methods during fine-tuning.

I propose to further extend my existing work, by applying these insights to a specialized case of mutation testing: generating tests that kill existing live mutants. I plan to include additional execution context into test generation models and use reinforcement learning. This will enable me to automatically generate test suites that are more comprehensive and similar to what an actual developer would write than current tools. I intend to complete this work by May 2025.

Contents

1 Introduction

Software testing is a critical component of the software development process. A high quality test suite can effectively find all inconsistencies between a system's requirements/specifications and its implementation; if a bug is introduced in a system, the test suite can detect the bug. These test suites both execute all code paths to ensure that all potential bugs are caught (high code coverage), and catch regressions in the code under test that a developer might introduce (high mutation score). However, writing high quality tests can be time-consuming [14, 15] and is often either partially or entirely neglected. This has led to extensive work in automated test generation, including both classical [12, 19, 30, 33] and neural-based methods [30, 98, 101].

Classical test generation tools like EvoSuite [33] directly optimize to generate high-coverage tests. However, the generated tests are often hard to read and may be unrealistic or even wrong [75]. This requires time and effort from developers to verify generated test correctness [19]. Fuzzing techniques [17, 106] face similar issues, pinpointing bugs in large scale systems, but also struggling with readability of automatically generated inputs. Meanwhile, language models trained on code have made major strides in generating human-like, high-quality functions based on their file-level context [13, 21, 34, 69]. Source code is natural [41], enabling language models to learn common, routine patterns present in source code. Tools like Copilot excel at code generation, and can significantly improve the productivity of its users [5], and offer some promising initial results in software testing.

However, these approaches applying language models to testing are limited. To generate a test, a developer must understand the code under test, including how to setup the necessary objects, invoke the method under test, and check some property about the code under test. Language models struggle with hallucination (invoking methods that are not in the source file). They also often fail to invoke internal methods present in the file under test, but not widely available and documented on the internet. This is because language models are pretrained on open source code and only 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. There is also a tight coupling between source and test files. Test files consist of individual test cases, that validate properties of source code. Without the source file, it is impossible to generate tests. Existing tests in the test file and project provide valuable information on the testing framework and structure of tests, thus should be used when they exist. Furthermore, there is a much stronger oracle for the quality of tests through execution feedback (compilation, passing, and coverage). This motivates the need to combine the existing local context that language models use with distant context in form of source method code and execution data.

```java
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 1.1: Unit test generation

```java
 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: Predictive 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.

I explore this insight in two software testing tasks: unit test generation and mutation testing. Figure 1.3 shows both the unit test generation and mutation testing tasks. Given a partially complete 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 predict whether a test suite can detect synthetic bugs (mutants). For the 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, 69], resulting in test generation models not considering the relationship between code and tests. As Figure 1.3 shows, it is impossible for a developer to generate the correct unit test without the code under test. Similarly, prior work in applying language models to mutation testing took limited context [56, 107] 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 1.3, one needs the test method body.

I propose leveraging the joint relationship between code and tests to improve the application of language models to software testing, both in pretraining and fine-tuning. In work, which is already completed, 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. I also show that this joint relationship between code and tests can be used to enhance state-of-the-art predictive mutation testing techniques, where I model mutants as a token level diff, and present the model with both the source and test method during fine-tuning.

I propose to further apply this insight to a specialized case of mutation testing: generating tests that are capable of detecting a bug that was previously undetected by the test suite. I plan to add additional execution context to existing test generation models. I will use reinforcement learning, with a policy that combines whether generated tests contain asserts, compile, pass, cover the mutated line of code, and ultimately kill the live mutant.

I will evaluate my work using a combination of metrics (some not evaluated in prior work) that are practically meaningful to developers. For unit test generation and test suite generation, I examine runtime metrics such as the compilation, passing the test suite, and coverage of generated test in addition to the commonly studied lexical metrics of CodeBLEU [82], and ROUGE [61] score. For mutation testing, I consider the execution time in a setting that guarantees no false positives and predictive power over non-trivial mutants. My techniques for unit test and test suite generation will be successful if they can generate tests that both look similar to developer written tests (as quantified by lexical metrics such as CodeBLEU and ROUGE), while also successfully compiling, passsing and improving code coverage (as quantified by runtime metrics). While I think this an improvement in lexical metrics will lead to more readable/usable tests and have some evidence to support this (a case study comparing EvoSuite to generated tests), a full human study is out of the scope of this thesis. Similarly, my techniques for mutation testing will be successful if they are capable of saving developer time even in a setting where the developer ensures there are no false positives.

I intend to complete this work by May 2025.

1.1 Thesis Statement

My thesis statement is:

We can exploit the structure of test code and close relationship between code and test files to enable the practical application of language models to software testing in both pretraining and fine-tuning. This approach can be used to a) generate useful unit test cases b) identify weaknesses in existing test suites and c) improve test suites to overcome found weaknesses.

1.2 Contributions

I propose a set of techniques that all exploit tests relationship with source code and execution data, not considered in prior work. My first two projects prove that this source context is helpful in both pretraining and fine-tuning software testing models. My final contribution adds an additional layer of execution data, that allows for generated tests to detect undetected sythetic bugs.

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 method context.

3. It demonstrates the effectiveness of adding execution context to test generation models, which enables us to generate mutant killing tests.
4. It evaluates all provided techniques with metrics and experiments that are practically meaningful developers, not considered in prior work

Although we focus on software testing, we believe that this insight extends to other domains. language models can leverage similar API specifications to generate examples of API parameters, improving API understanding [11]. Property testing of widely used libraries can leverage their uniquely thorough documentation to improve correctness and quality of language model generations [97]. Applying language models to new domains should both look at the structure of domain specific tasks, and leverage domain-specific information.

1.3 Evaluation Metrics

A contribution of my thesis is evaluating software testing techniques in a meaningful way. Prior work on language model unit test and test suite generation [68, 92, 93] evaluated their approaches on lexical metrics such as CodeBLEU [82] and ROUGE [61] score. These metrics only quantify how close do generated tests look to developer written tests. However, they do not correlate well to practical utility; a test that does not compile or check interesting properties can still have very high CodeBLEU and ROUGE scores. Furthermore, a test that checks an interesting property, but looks syntactically different from developer tests, could be practically useful, but have very low CodeBLEU and ROUGE scores. In my work [81], 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. These metrics closely align with the end goal of automated test generation approaches: to generate meaningful tests that cover new code paths and catch potential bugs. I plan to use this combination of lexical and runtime metrics in my proposed work, along with adding an additional metric for readability of generated tests [25], as we claim that the language model generated will be more readable than tests generated by search based approaches.

For the mutation testing task, I add to the evaluation of existing work [56, 107] by considering a setting where the tool checks all predicted undetected mutants to avoid showing the developer false positives. Multiple studies [49, 67] 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. 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.

1.4 Proposal Outline

The rest of the proposal is structured as follows. In Chapter 2 I discuss the related work in test generation, mutation testing, machine learning, and a review of the literature. Next, in Chapter 3, Chapter 4, and Chapter 5 I discuss three research projects including existing and proposed work. In Chapter 6, I outline my proposed timeline and I conclude in Chapter 7.

2 Related Work

2.1 Test Generation

There is extensive work in automated test generation, falling in two categories: classical test generation and neural test generation. Classical generation employs software engineering techniques such as fuzz testing and genetic programming to generate test suites, while neural test generation techniques employ machine learning techniques. Both techniques have their pros and cons: classical techniques tend to produce test suites with high coverage, but struggle with readability, while neural test generation techniques have high readability, but struggle to achieve coverage goals.

2.1.1 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 [74], aflplusplus [32] and honggfuzz use coverage to guide generation of test prefixes. Property testing tools such as Korat [18], QuickCheck [23] and Hypothesis [65] allow a developer to specify a set of properties and subsequently generates a suite of tests that test the specified properties. PeX [90] and Eclipser [22] 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 [30], which limits the level of testing they provide. EvoSuite [33] 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 [19, 26, 83].

2.1.2 Neural Test Generation:

More recently, neural test generation methods have been developed to generate more natural and human understandable tests. ConTest[98] makes use of a generic transformer model, using the tree representation of code to generate assert statements. ATLAS [101], ReAssert [103], AthenaTest [92] and TOGA [30] 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 [68] 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.

2.2 Mutation Testing

Mutation testing [29] 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. These operators vary [24, 36, 50], 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. The change in Figure 4.1a creates one such mutant for the `next()` method.

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 [52, 77]. Mutation testing has seen some industry adoption [16, 78]. Prominent recent uses at Facebook and Google apply it only to changed code at commit-time, which still requires large amounts of idle compute [79] because of the massive computational expense of running it over an entire codebase.

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 [54, 71, 94, 107]. Approaches to reducing the cost of mutation analysis were categorized as *do smarter*, *do faster*, and *do fewer* by Offutt and Untch [70]. 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 [56, 66, 107] 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 [107] 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 [56] achieves higher accuracy with lower overhead by exclusively using information about the source code and mutation itself (source method, test method, and mutated line).

2.3 Language Models of Code

Language models can perform well across many tasks when prompted with instructions and examples [20, 91]. Codex [21] is an autoregressive (left to right generation) language model with 12B parameters, fine-tuned from GPT-3 on 54 million GitHub Python repositories. CodeGen-16B, with which we compare, outperforms this model [69]. Later, unpublished, iterations of Codex have also been applied to commercial settings, powering GitHub's Copilot [5]. TestPilot [85] uses Codex to generate unit tests. However, it requires significant volumes of documentation as input, which is often not available for open-source projects. More recently, new models such as DeepSeekCoder [38] and CodeLlama [84] show promising code generation abilities. These models support both autoregressive text completion and text infilling, with large context windows enabled by sparse attention.

Closed source models [72, 89] have also shown promise with massive context windows ranging from 128k to 1 million tokens and stronger code generation abilities than their open source counterparts. These models can ingest hundreds of pages of documentation and entire code repositories, making them have the best code completion abilities. However, a limitation is that their closed source nature makes them impractical to many enterprises who do not want to risk data leakage. Additionally, due to their closed source nature, such models have a (relatively) high cost per request, also making them impractical as the number of requests scales up.

While all of these models perform well at generating code, they are relatively poor (for their size) at generating *tests* for the code. These models are typically trained on a randomly shuffled corpus of entire files, and do not learn the alignment of tests to the code under test, therefore struggle with common issues such as hallucination and failing to invoke internal API methods. While large corporate models perform (relatively) better in these dimensions, their closed source nature inherently limits their use cases.

3 Training Language Models on Aligned Code And Tests

In this chapter, I leverage the close relationship between code and test files to improve unit test generation.[1] As discussed earlier, generating unit tests is a challenging and time consuming task, where automation has potential to add significant value. For a developer to generate unit tests they must consider both the code file and test file. Thus it is not surprising that existing test generation approaches [5, 30, 68] 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.

My collaborators and I 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 strong 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 [60] and, the CodeGen [69] model family, which includes mono-lingual models trained on a much larger budget than ours. We also compare against TeCo [68], 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.

[1]Completed work that appeared in ASE 2023 [81]

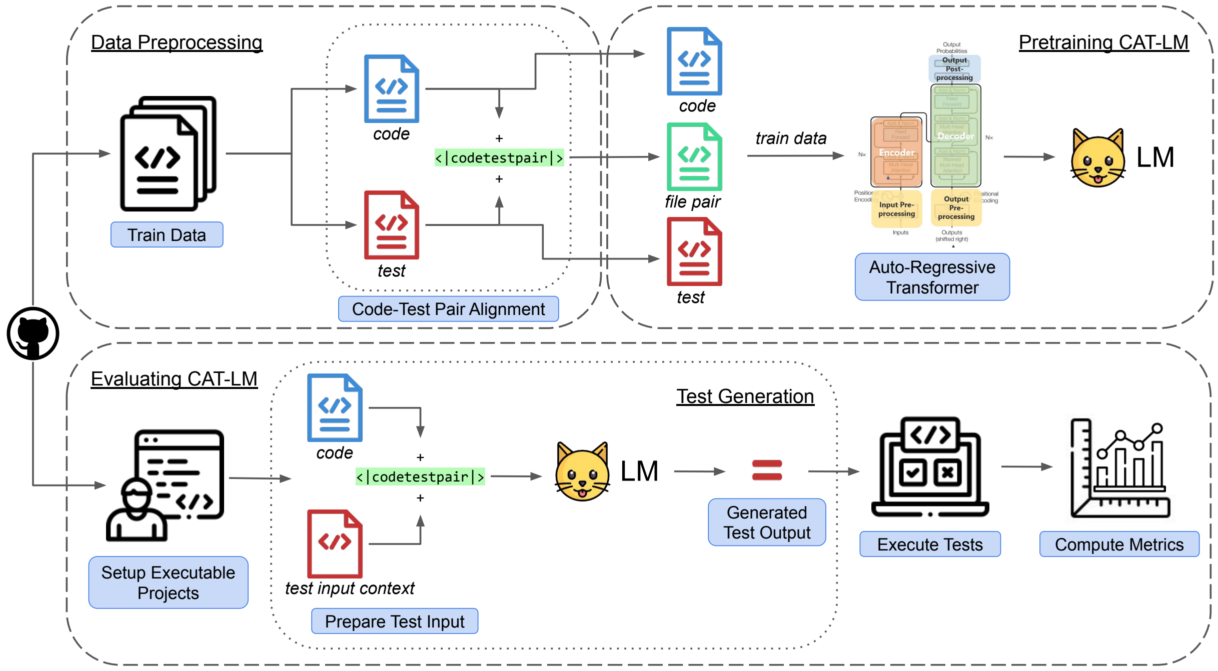


Figure 3.1: Approach overview. We extract Java and Python projects with tests from GitHub and heuristically align code and test files (top), which, along with unaligned files, train CAT-LM, a large, auto-regressive language model. We evaluate CAT-LM's generated tests on a suite of executable projects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

3.1 CAT-LM 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 ca. 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 [102] 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 [42]. We use the GPT-NeoX toolkit [2] enhanced with Flash Attention [28]

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.

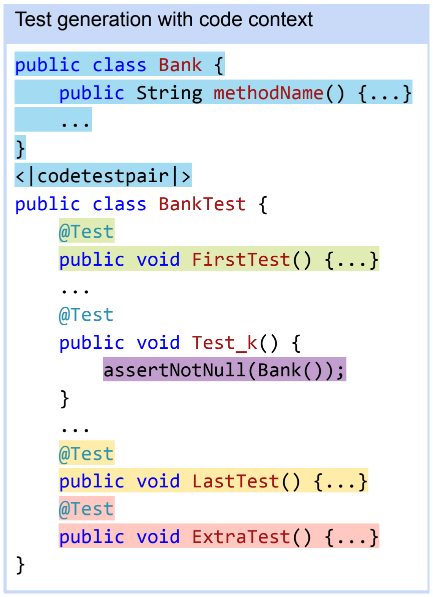
Finally, we evaluate CAT-LM on the held-out test data. We manually set up all projects with executable test suites from the test set to form our testing framework. We prepare our test inputs for CAT-LM by concatenating the code context to the respective test context for test generation. The test context varies based on the task. We asses 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.

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



```
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.

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.

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 4.1 provides high-level statistics pertaining to data collection and filtering.

Table 3.1: Summary statistics of the overall dataset.

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

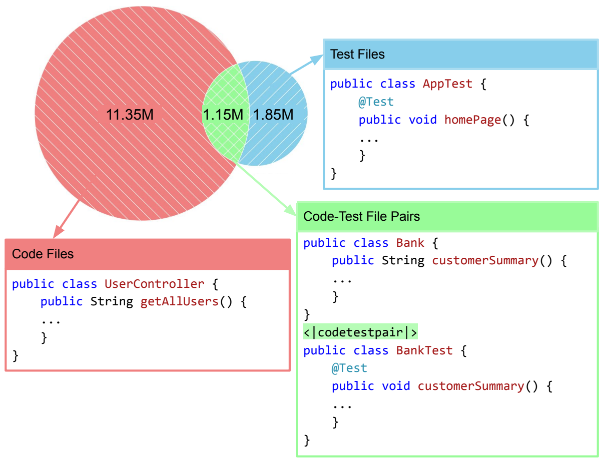
3.3.1 Data Collection

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 [62], 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 ∼23M files (about 280 GB). We randomly split this into 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 [102] 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.



```
public class AppTest {
    @Test
    public void homePage() {
    ...
    }
}
```

```
public class Bank {
    public String customerSummary() {
    ...
    }
}
<|codetestpair|>
public class BankTest {
    @Test
    public void customerSummary() {
    ...
    }
}
```

```
public class UserController {
    public String getAllUsers() {
    ...
    }
}
```

Figure 3.3: Distribution of files with sample code snippets

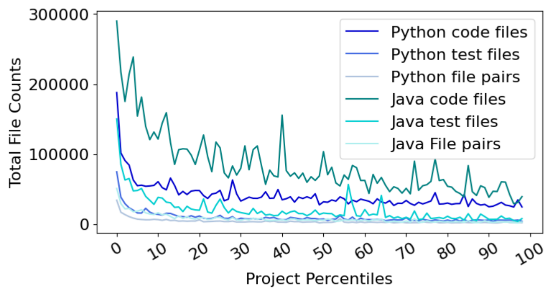


Figure 3.4: Distribution of files in projects sorted by GitHub stars, normalized by percentiles

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 [57, 87]. 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 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 [58] 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 current code generation models use a context window of up to 2,048 tokens [69, 104].[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

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

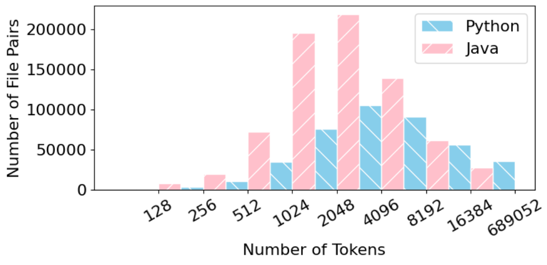


Figure 3.5: Distribution of file pair tokens

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 ∼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.

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 [42], 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 [69, 104], 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 [69, 91];[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

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

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.

3.5 Experimental Setup

We describe the setup for evaluating CAT-LM across both tasks outlined in Section 3.2, namely test method generation, and test completion. We extend prior evaluations of neural test generation approaches by adding runtime metrics to the standard lexical evaluation of tools. These runtime metrics correlate with the practical utility of generated tests; developers would likely 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 auto-regressively (left-to-right) [69]. 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 multi-lingual 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 [21] on a Python benchmark [69].

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 [60], 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 [13] 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 [61] (longest overlapping subsequence of tokens) and CodeBLEU [82] 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 [43, 59, 68, 100].

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 [30, 68, 101]. For additional detailed descriptions of all lexical and run-time metrics, results are available in published work [81].

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 [81].

Testing Framework

We evaluate the quality of the generated tests using the containers that we setup 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.

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

PL	Case	Cov Imp %	# 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

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.

Baseline Model

We compare against TeCo [68], a state of the art baseline on test statement completion that has outperformed many existing models, including CodeT5 [100], CodeGPT [63] and TOGA [30]. TeCo [68] is a encoder-decoder transformer model based on the CodeT5 architecture [100]. 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.

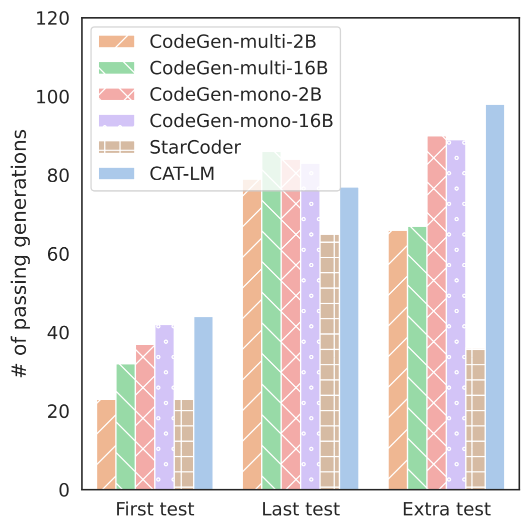
Initially, we intended to compare CAT-LM against TeCo on our test set. However, TeCo performs extensive filtering including requiring JUnit, Maven, well-named tests, a one-to-one mapping between test and method under test, and no if statements or non-sequential control flow in the test method. We thus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

Metrics

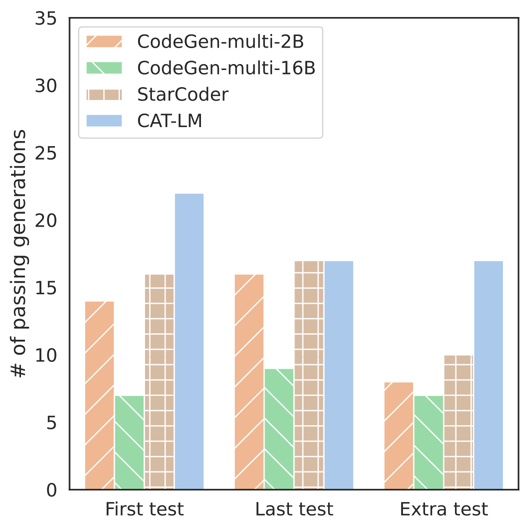
We compare CAT-LM against TeCo across all lexical metrics (outlined in Section 3.5.1).

3.6 Evaluation

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. Additional results can be found in published work [81].



(a) Python.



(b) Java.

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

3.6.1 Test Method Generation

Pass Rate

Figure 3.8 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 don't show significant gains with more parameters or longer contexts (StarCoder can use $8,192$ tokens), highlighting that training with code context is important.

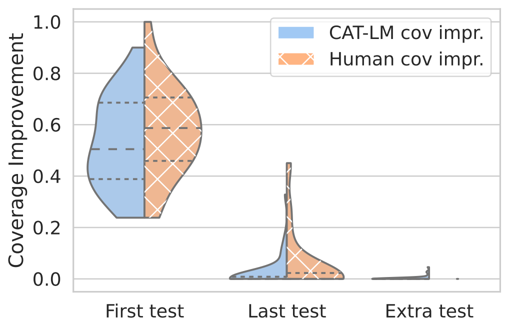
[4]The denominator for each group is the number of file pairs shown in Table 3.2 multiplied by 10, the number of samples per context.

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

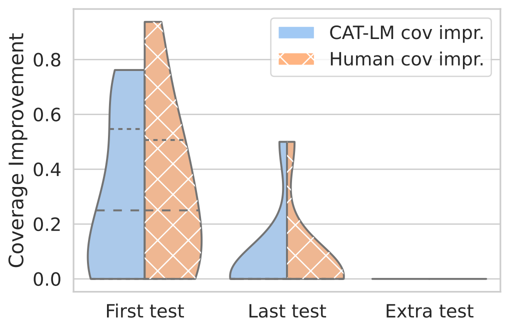
	Lexical Metrics			**Runtime Metrics**	
Model	**CodeBLEU**	**XMatch**	**Rouge**	**Compile**	**Pass**
First Test (Total: Java = 270)					
CAT-LM w Context	41.4%	15.4%	60.9%	50	22
CAT-LM w/o Context	37.5%	15.4%	56.5%	9	9
Codegen-2B	35.5%	7.7%	56.8%	24	14
Codegen-16B	42.2%	7.7%	61.8%	25	7
StarCoder	44.6%	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) 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.

Coverage

Figure 3.9 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 3.8), 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).

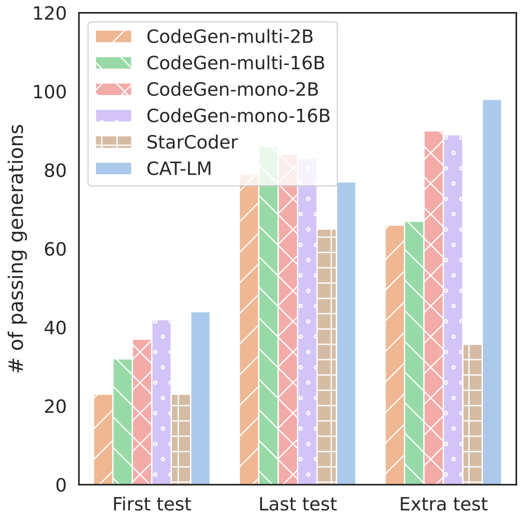
We note that we could not compute coverage for all the file pairs in each setting. We excluded file pairs with only one test from our last test setting to differentiate it from our first test setting. For the first test setting, some baseline files were missing helper methods between the first test and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

Lexical Similarity

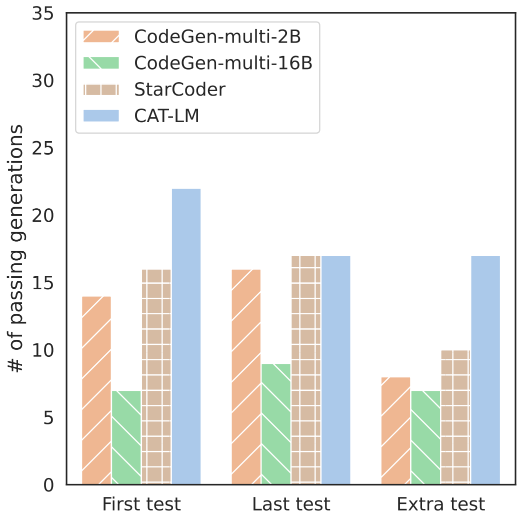
Table 3.4 and Table 3.3 show the lexical similarity metrics results relative to the human-written tests for CAT-LM, both with and without context, along with StarCoder and CodeGen baselines. CAT-LM reports high lexical similarity scores when leveraging code context, typically at or above the level of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

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



(a) Python.



(b) Java.

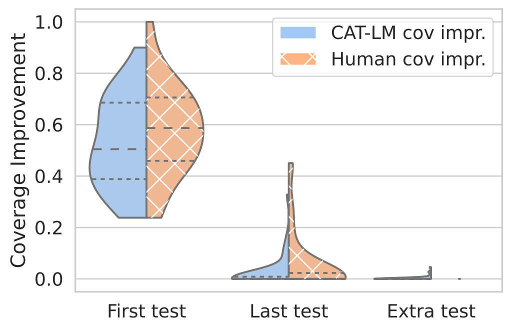
Figure 3.8: Passing tests by model for Python and Java.

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.

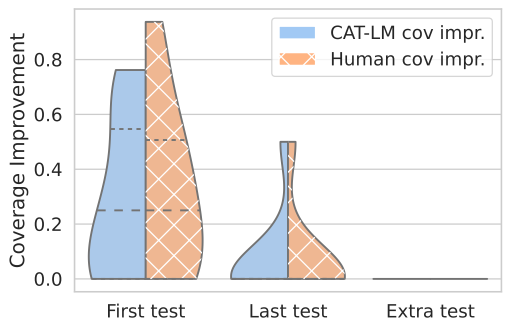
In terms of runtime metrics, code context mainly helps on the first and last test prediction task, with especially large gains on the former. Context does not seem to help generate more passing tests in the extra test setting. This may be in part because the test suite is already comprehensive, so the 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.



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



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

Figure 3.9: Coverage improvement of our model vs humans for different languages.

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%

3.6.2 Test Completion

For test completion (see Section 3.2.2 for task definition), we compare CAT-LM against TeCo [68] 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.

3.7 Limitations and Threats

Limitations: One limitation of CAT-LM is our use of flash attention [28]. 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.

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. This coupling between code and test files can also help improve other software testing tasks such as mutation testing (chapters Chapter 4 and Chapter 5).

4 Contextual Predictive Mutation Testing

In this chapter, I apply the key insight that test code and source code are tightly coupled to automatically detect inadequacies in exisiting test suites *without* executing the tests.[1] Recall, the goal of mutation testing is to find synthetic bugs (mutants) that existing tests fail to detect. The main limitation of mutation testing is that for each synthetic bug introduced, the entire test suite needs to be run, making it costly to scale. We can overcome this problem 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), and significantly reduce test execution time.

While this technique is prpmising, prior work in predictive mutation testing [56, 107], took limited context such as the test method name and mutated line and thus failed to achieve performance needed for practical use. We leverage my thesis insight that there is a 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 [96] architecture to encode source and test methods, and further finetune it for our task.

We evaluate MutationBERT in both same project and cross project settings, measuring both accuracy and execution time. Thanks to its higher precision, MutationBERT saves 33% 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.

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.

[1]Completed work that appeared in FSE 2023 [45]

```
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 marked in brown.

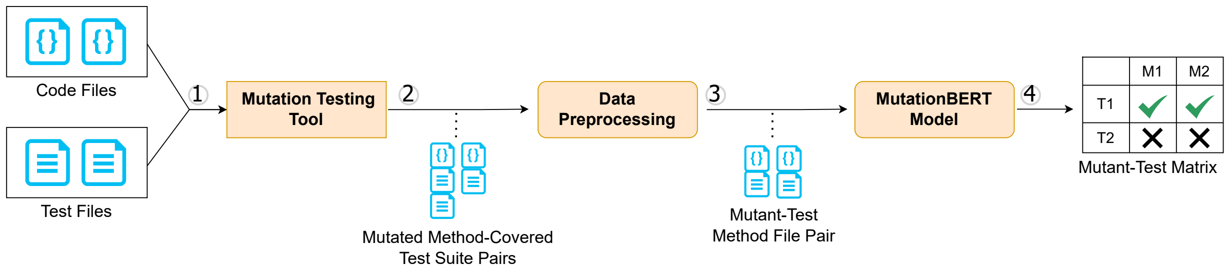


Figure 4.2: An overview of MutationBERT's workflow. Step ① provides source and test files to a mutation testing tool. In Step ②, the mutation tool generates mutants and correspondng covering tests, which are preprocessed, tokenized, and formatted. In Step ③, MutationBERT takes these inputs to produce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

4.1 Contextual Predictive Mutation Testing

Figure 4.2 overviews the MutationBERT workflow. Our workflow takes a project and test suite as input, and uses a given source-level mutation testing tool (step ① to generate 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 [16, 54, 78]. 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 a given `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 [56, 107] representing the modification with a graph [64, 99, 105] or representing the "before" and "after" of the modification with multiple embeddings [88].

For earlier PMT models [56, 107] 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 [31], 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 [31] 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 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

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

to the mutation testing task, with a novel input representation. CodeBERT [31] is a pretrained model that leverages the transformer architecture [96]. 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 [31], to merge conflict resolution [88], 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:

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

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 [56], 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 [45]. Our evaluation extends prior work [56, 107] 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).

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.

[3]Full details can be found in published work [45]

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.

RQ3: 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.

RQ4: 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.

RQ5: 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.

4.2.1 Baseline

We compare against the Seshat baseline [56]. Seshat is a state-of-the-art model for mutation testing, which has been shown to outperform PMT [107] 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 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 concatinates 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 a threshold in their paper, so we perform the same optimization as we did for MutationBERT).[4] We thus aggregate as follows:

$$\texttt{pred}_{M,T} = \begin{cases} \text{``detected''} & (max_{t \in T} Seshat\,(M,t)) > 0.10 \\ \text{``undetected''} & \text{otherwise} \end{cases} \tag{4.2}$$

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

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

4.2.2 Dataset

We reuse the dataset released with the Seshat experiments [56]. This dataset consists of a full mutation analysis in Major [50] 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 [56] 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. [56].

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 a 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 [86]) with vocabulary size of 50,000 tokens for all programming languages that is 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. [56] 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 warmup to 1000 steps, followed by a cosine annealing decay, in accordance with best practices for fine-tuning transformers [80]. Both models' loss functions converge using these settings. We fine-tuned our model on a 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 mutants. That is, a test detecting a mutant is the minority class. In this setting, model *precision*

[4]Full details on this comparison can be found in published work [45]

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 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, 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 RQ4, 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 warmup 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 [56] in conjunction with our inference time numbers.

For RQ5, 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 Results and Analysis

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

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.

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**

4.3.1 RQ1: Same Project Performance

Table 4.3 shows the results of MutationBERT and Seshat on the test set for the *same 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. 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 (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 [52, 77] 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.

4.3.2 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,

Table 4.4: 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

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, 40]. 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, 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).

4.3.3 RQ3: 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 4.4. 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).

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 mutant detetion. 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.

Table 4.5: 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

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

```
public BufferedImage createBufferedImage(..., ChartRenderingInfo info) {
  ...
- if (info != null) {
+ if (true) {
    info.setRenderingSource(...);
  }
}

public void testDrawWithNullInfo()

```

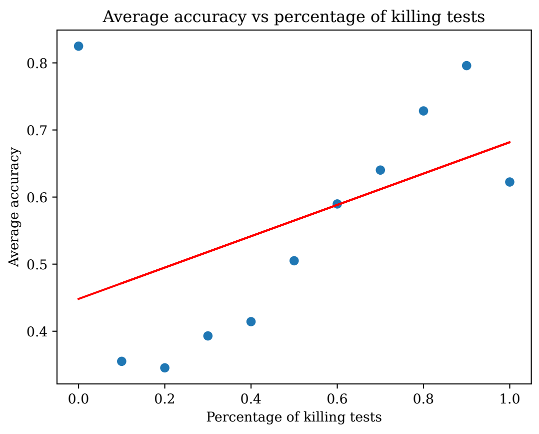
This example shows a mutant that replaces a null check on `info` with `true`. Since the test is a case where `info` is null, on the mutated code, there will be a null pointer dereference. Thus a `NullPointerException` will be thrown and the mutant will be killed. MutationBERT fails to see the correlation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

4.3.4 RQ4: Efficiency

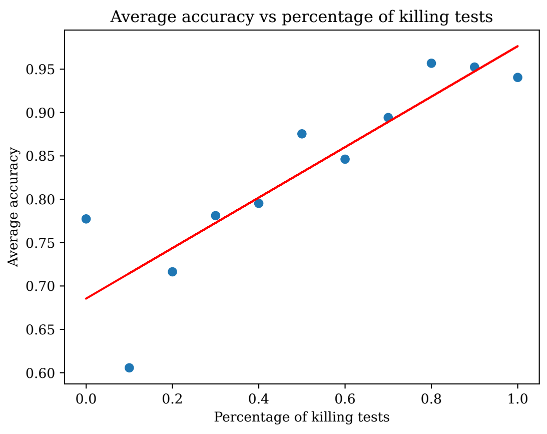
Finally, we discuss the efficiency and performance benefits of MutationBERT as compared to Major or Seshat. Table 4.5 shows time to run each tool, including Major, for all mutants in a project (center column), and time to run including a confirmatory check for the predictive techniques (right-hand columns).

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 4.5 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



(a) Seshat



(b) MutationBERT

Figure 4.3: Accuracy vs. percentage of killing mutants for Seshat and MutationBERT

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 4.5 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.

4.3.5 RQ5: Mutant Importance

Figure 4.3 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.

4.4 Limitations and Threats

Limitations: MutationBERT depends on GPU availablity 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). 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 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 external threats to validity lie in our dataset of mutants and tests. 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. We also practically discuss how these metrics translate to the real world use case.

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 [81] 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.

Adiditionally, MutationBERT is *practically* useful for both of the core end user tasks in mutation testing: 1) as a more complete measure of testing adaquacy (computing mutation score) [35, 71] and 2) to identify undetected mutants that indicate potential inadequacies in existing testing efforts [16, 78].

In the classical sense, mutation testing serves to evaluate test suite quality [29, 39, 47]. 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,

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

substantially lower than Seshat. Further note that as Table 4.5 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 [35], 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 [78] and Facebook [16] use mutation testing to pinpoint undetected mutants that reveal issues with test adaquacy. MutationBERT substantially saves time here, as unlike Seshat, it still achieves over 60% accuracy in predicting hard to detect mutants. When shown a set of undetected mutants, a developer would be able to trust MutationBERT's output. 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 takes 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 Generating Mutant Killing Test Suites

In this chapter, I will present proposed work that combines the tight relationship between code and tests, with novel execution data to generate mutant killing suites. One of mutation testing's major limitations is the human cost of acting on undetected mutants. Large-scale systems commonly have hundreds of thousands of mutants [35, 37], since mutants scale with size of the codebase and mutation operators considered, resulting in thousands of undetected mutants. For each undetected mutant, a developer needs to manually examine the source code changed, and (ideally) generate a test that is capable of catching and killing the mutant.

I hope to automate this process, by automatically generating mutant killing tests for a subset of these cases by leveraging language models in combination with execution metrics, such as the generated test compiling, passing, covering the target line of code and ultimately detecting the mutant. The resulting tool would be similar to EvoSuite [33], which also generates mutant killing tests, yet considering the context in which LLM operate, we trust the tests would be what one may consider "more natural". This is different than Dakhel et al. [27], as we target only open source models and more complex targets, where generating tests is not trivial (as in HumanEval [21]).

I aim to target tests that cover the mutated line and lead to different output from the original program such as to kill the mutant. I plan to combine Reinforcement Learning (RL) techniques with existing test generation models to accomplish this task. Unlike code generation, there are much stronger signals in testing to guide RL (whether the test has assert statements, compiles, passes, covers the mutated line of code, leads to dissimilar coverage from the original program, produces different output and finally kills the mutant). I aim to generalize and show the potential of the approach in both Python and Java.

5.1 Approach

We outline our high level modeling approach, including model inputs and outputs and the modeling pipeline.

5.1.1 Input Representation

We plan to feed the model two pieces of information: the mutant encoded with additional tokens `<BEGIN>`, `<MUTANT>`, and `<END>` and a test that covers the mutated line of code, but does not detect the mutant. This is in line with published work outlined in Chapter 4. Below is an example of an input we would feed to the model and the expected model output.

Model Inputs: We provide the model with both mutation encoded as a token diff and a covering test.

Mutation:

```python
def foo():
if a <BEGIN>><MUTANT>>=<END> 10:
  return True
return False
```

Covering Test:

```python
def t1():
 assert(foo(11) == True)
```

Model Output: The output is a mutant killing test.

Mutant Killing Test:

```python
def t2():
    assert(foo(10) == False)
```

5.1.2 Approach

We plan to fine tune a code generation model to the task of test generation to get good at generating natural tests that are capable of detecting mutants. State-of-the-art open source models that we can feasably fine-tune for testing tasks include CodeLlama-7B [84], Mistral-7B [48], and DeepSeekCoder [38]. We plan to take our dataset of mutants and tests and obtain tests that cover a mutant but do not detect it along with the "gold" detecting test. We supply a mutant and covering test as input and fine-tune the model to generate the detecting test.

We then plan to run reinforcement learning on the fine tuned test generation model This requires defining a policy that takes into account compilation, passing, coverage (branch coverage), and whether the mutant is killed or not. Unlike code generation, we have these strong signals as to whether a test is "good" or not. In line with reinforcement learning with human feedback [73], we plan to further fine-tune a subset of parameters in the test generation model with the execution policy. Here we can either use a manually specified reward function with these execution signals, or use a language model to learn the reward function, which would allow us avoid running these metrics on all generated tests. Instead we would run execution metrics on a subset of generated tests and measure whether the generated test detects the mutant. Then we would use this data to train a reward model, which we would use in the reinforcement learning loop.

5.2 Evaluation

We describe the dataset, baselines and metrics used for our evaluation of our approach against baselines.

Research Questions

RQ1: Mutant Coverage: What proportion of mutants can mutant LLM test suite generation generate detecting tests for? What is the code coverage of the final test suite? We measure the proportion of mutants that we can generate detecting tests for, and the code coverage of the final test suite. We also report runtime metrics (proportion that compile, pass the test suite) for the generated tests.

RQ2: Test Readability: How readable are LLM test suite generated tests? How similar are LLM test suite generated tests to developer tests? We measure readability in line with prior work [25], and compare the generated tests to developer written tests using lexical similarity metrics. The goal of this RQ is to answer whether generated tests are more readable than EvoSuite [33] generated tests.

RQ3: Qualitative Study: What kinds of mutants is LLM test suite generation not able to generate detecting tests for? We manually inspect a subset of mutants that we are not able to generate detecting tests for, and provide a qualitative analysis of why the model fails to generate a detecting test. We hope to provide insights into the limitations of the model and areas for future work.

5.2.1 Dataset

We plan to use Defects4J [51] and a hand-created dataset of recent projects that LLMs would not see at pretraining time for measuring LLM performance. Defects4J consists of approximately 85k mutants, which can be used for training, namely given a mutant generate a mutant killing test. Since the LLM has seen this dataset a pretraining time, we acknowledge of possibility of data leakage in evaluation, thus only use this dataset for evaluation on baselines that we cannot compare on our other dataset. We also plan to create a dataset of recent projects that we can run mutation testing. We would filter out projects that do not use JUnit and unittest to ensure that our model has seen relevant test framework code during fine-tuning. We plan to use this dataset for the bulk of our evaluation, as it fully mitigates data leakage threats. To measure performance on undetected mutants, we plan to PR generated tests and measure developer acceptance of generated tests.

5.2.2 Baselines

EvoSuite: EvoSuite [33] is a search-based test generation approach. It uses genetic programming to generate tests capable of killing mutants, however struggles from readability problems, with tests not looking similar to ones generated by developers.
CAT-LM: CAT-LM [81] is a state-of-the-art test generation approach. We plan to measure the coverage and mutation score generated by CAT-LM and our approach.
MuTAP: MuTAP [27] is a state-of-the-art test suite generation approach that leverages prompting, rather than the reinforcement learning technique we propose. We plan compare lexical, runtime metrics along with the readability of tests generated by both tools.
GPT-4: GPT-4 is a closed-source large language model [72], with state-of-the-art performance on a wide variety of tasks. Since GPT-4 is significantly larger than any model we plan to fine-tune, we expect it to outperform our model. We plan to use GPT-4 as an upper bound on performance, to show the potential of our approach.

5.2.3 Metrics

We define the following runtime and lexical metrics for comparing against existing baselines. Similar to CAT-LM, we argue that including these runtime metrics is essential for evaluating the practical utility of generated tests. We also plan to include an additional readability metric [25], as we hypothesize language model generated tests will be more readable than existing search-based baselines.

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 and mutation score of the generated tests. We also used these runtime metrics in evaluating CAT-LM [81].

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 [61] (longest overlapping subsequence of tokens) and CodeBLEU [82] 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 generation [43, 59, 68, 81, 100].

Additionally, to better understand the readability of generated tests, we plan to use the regression model from Daka et al. [25]. This model is based off of human annotations of readability of EvoSuite generated tests. It factors in metrics such as the number of lines, the byte entropy of statements, and the number of loops present in a given test.

6 Proposed Timeline and Risks

6.1 Timeline

I propose the following timeline with an expected defense date in May 2025. At the time of writing this proposal I have published the study described in Chapter 3 at ASE 2023 and Chapter 4 at FSE 2023. I have begun exploratory work on the study described in Chapter 5, but still need to design experiments and prepare a publication.

- **January-May 2024**
  - Complete the proposal document and propose my thesis.
- **June-October 2024**
  - Mine GitHub for a dataset of Java and Python projects.
  - Execute tests and run mutation testing to get a dataset.
- **October-November 2024**
  - Train a mutant test generation model.
  - Conduct initial evaluation, Defects4J.
- **November-December 2024**
  - Conduct test generataion experiments and compare to existing work.
  - Prepare a paper for submission to a software engineering venue.
- **January-April 2025**
  - Finish work in progress, including any papers under revision.
  - Write thesis document.
- **May 2025**
  - Defend and graduate.

6.2 Risks

One major risk is using reinforcement learning as a part of my proposed work. Reinforcement learning is known to be unstable, however I think the more deterministic nature of metrics used in our reinforcement learning loop (coverage, mutant killing behavior) will help mitigate this risk. In addition to traditional RL, I also plan to explore offline approaches and fine-tuning without this execution data, which may work better due to increased stability.

Another risk with this project is that open source test generation models might not be powerful enough to generate entire test suites, with current state of the art open source models significantly lagging behind closed source models such as Gemini and GPT-4, which are both capable of generating, stronger, high quality tests. I hope that my work can help bridge this gap, creating open source test generation models that companies can integrate into their workflow. In the current setup, there are many developers who cannot use closed source models, due to data leakage and privacy concerns. The cost per request is also significantly less with these (relatively) small models, making them much more scalable than large models such as GPT-4.

6.3 Stretch Goal (or Backup Plan): Predicting Equivalent Mutants using LLMs

As a stretch goal or backup plan in the case that my proposed work does not work out due to risks outlined above, I plan to leverage the strong code understanding of LLMs to predict whether mutants are equivalent or not. A major limitation of mutation testing are equivalent mutants, which are mutants that are semantically equivalent to the original code under test. These mutants will always be undetected, being flagged as cases that a developer should look at when refining their test suite, despite the fact that they do not reveal inadequacies in the test suite. This is problematic, as equivalent mutants waste developer time, making mutation testing a less practical test refinement technique. This is also a limitation of my prior work on predictive mutation testing, as equivalent mutants are likely to be predicted as undetected, meaning these approaches would also have these false positive cases.

LLMs have shown promise in understanding code semantics, with recent work showing that LLMs can predict whether code snippets are semantically equivalent or not [6, 72]. There has been some work in code clone detection [44, 46], but primarily using LSTMs and smaller language models. Furthermore, no one has applied these clone detection techniques to equivalent mutants tasks. A big challenge is that there is no large scale dataset of equivalent mutants, with most existing datasets only containing a few programs (for example the triangles benchmark, with only one problem). Thus the contributions of this work are as follows:

1. We introduce a dataset of equivalent mutants from a source level mutation testing tool, where humans manually labeled mutants as equivalent and non-equivalent. We release this dataset for future research.
2. We benchmark multiple LLMs, prompting approaches, and prior approaches to predict equivalent mutants, measuring performance on our dataset.

To accomplish this task, we plan to explore multiple different approaches, including prompting state-of-the-art LLMs, and prior code clone detection approaches. We also plan to incorporate execution data as part of the input to these models, such as whether the mutant is covered by the same set of tests, along with an execution trace of the mutant. Even if our final approach requires running the existing test suite on the mutant, we believe filtering out these false positive cases outweighs the cost of running the test suite on the mutant. We plan to evaluate our approach on two separate benchmarks. Firstly, we have a dataset of equivalent mutants from prior work, [1] across Rust, C++, Python, and Java from highly starred projects. We also plan to compare against MutantBench [95], a benchmark of multiple introductory programs and equivalent mutants. We plan to compare against multiple baselines, including TCAP (a technique for mutant prioritization) [55], trivial compiler equivalence (checking compiled programs for equivalence) [76], and state infection approaches [53]. Overall, we hope that we can provide a practical tool to filter out equivalent mutants, making mutation testing more practical for developers.

[1]https://agroce.github.io/fse24.pdf

7 Conclusion

Software testing is fundamentally different than 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 propose bootstrapping language models with execution context for the task of mutant test generation, where we will use reinforcement learning on common execution metrics of coverage and mutant killing. Although this proposal primarily focuses on software testing, I believe that this insight extends to other domains, as demonstrated by recently published work on API understanding and property testing [11, 97].

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, 2.3, 3

[6] ChatGPT, November 2022. URL `https://openai.com/blog/chatgpt/`. 1, 6.3

[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

[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. 4.3.2

[10] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, SPLASH '19, pages 143–153, 2019. 3.3.1

[11] Juan C. Alonso, Alberto Martin-Lopez, Sergio Segura, José María García, and Antonio Ruiz-Cortés. Arte: Automated generation of realistic test inputs for web apis. *IEEE Transactions on Software Engineering*, 49(1):348–363, 2023. doi: 10.1109/TSE.2022.3150618. 1.2, 7

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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. 2.2, 4.1, 4.5

[17] Marcel Böhme and Brandon Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 713–724, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409729. URL `https://doi.org/10.1145/3368089.3409729`. 1

[18] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. *SIGSOFT Software Engineering Notes*, 27(4):123–133, 2002. 2.1.1

[19] 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, 2.1.1

[20] 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. 2.3

[21] 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, 2.3, 3.5.1, 5

[22] 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. 2.1.1

[23] 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. 2.1.1

[24] 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. 2.2

[25] 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`. 1.3, 5.2, 5.2.3, 5.2.3

[26] 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. 2.1.1

[27] 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.

[28] 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, 3.7

[29] 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. 2.2, 4.5

[30] 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, 2.1.1, 2.1.2, 3, 3.5.1, 3.5.2

[31] 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

[32] 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. 2.1.1

[33] 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, 5, 5.2, 5.2.2

[34] 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

[35] 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, 5

[36] 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. 2.2

[37] 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. 5

[38] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. 2.3, 5.1.2

[39] 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

[40] 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. 4.3.2

[41] 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

[42] 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

[43] 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, 5.2.3

[44] Cheng Huang, Hui Zhou, Chunyang Ye, and Bingzhuo Li. Code clone detection based on event embedding and event dependency. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, Internetware '22, page 65–74, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450397803. doi: 10.1145/3545258.3545277. URL `https://doi.org/10.1145/3545258.3545277`. 6.3

[45] 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, 4.2, 3, 4

[46] Hajin Jang, Kyeongseok Yang, Geonwoo Lee, Yoonjong Na, Jeremy D. Seideman, Shoufu Luo, Heejo Lee, and Sven Dietrich. Quickbcc: Quick and scalable binary vulnerable code clone detection. In *IFIP International Information Security Conference*, 2021. URL `https://api.semanticscholar.org/CorpusID:235680083`. 6.3

[47] 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

[48] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. 5.1.2

[49] 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. 1.3

[50] 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. 2.2, 4.2.2

[51] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.2628055. URL `https://doi.org/10.1145/2610384.2628055`. 5.2.1

[52] 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 *Symposium on Foundations of Software Engineering*, FSE '14, pages 654–665, 2014. URL

https://doi.org/10.1145/2635868.2635929. 2.2, 4.3.1

[53] René Just, Michael D. Ernst, and Gordon Fraser. Using state infection conditions to detect equivalent mutants and speed up mutation analysis, 2013. 6.3

[54] 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, 4.1

[55] Samuel J. Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. Prioritizing mutants to guide mutation testing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1743–1754, 2022. doi: 10.1145/3510003.3510187. 6.3

[56] 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, 1.3, 2.2, 4, 4.1.1, 4.2, 4.2.1, 4.2.2, 4.2.3, 4.2.4

[57] 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

[58] 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

[59] 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, 5.2.3

[60] 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

[61] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Conference on Text Summarization Branches Out*, pages 74–81, 2004. 1, 1.3, 3.5.1, 5.2.3

[62] 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

[63] 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

[64] 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

[65] 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. 2.1.1

[66] 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

[67] 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. 1.3

[68] 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. 1.3, 2.1.2, 3, 3.5.1, 3.5.1, 3.5.2, 3.6.2, 5.2.3

[69] 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, 2.3, 3, 3.4.1, 3.4.2, 3.5.1

[70] 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

[71] 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, 4.5

[72] OpenAI. Gpt-4 technical report, 2023. 2.3, 5.2.2, 6.3

[73] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022. 5.1.2

[74] 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. 2.1.1

[75] 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

[76] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 936–946, 2015. doi: 10.1109/ICSE.2015.103. 6.3

[77] 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. 2.2, 4.3.1

[78] 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. 2.2, 4.1, 4.5

[79] 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. 2.2

[80] 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

[81] N. Rao, K. Jain, U. Alon, C. Goues, and V. J. 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`. 1.3, 1, 3.5.1, 3.5.1, 3.6, 4.5, 5.2.2, 5.2.3, 5.2.3

[82] 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. *CoRR*, abs/2009.10297, 2020. 1, 1.3, 3.5.1, 5.2.3

[83] 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. 2.1.1

[84] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron

Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. 2.3, 5.1.2

[85] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. Adaptive Test Generation Using a Large Language Model. *CoRR*, abs/2302.06527, 2023. 2.3

[86] 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

[87] 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

[88] 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

[89] 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. 2.3

[90] 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. 2.1.1

[91] 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. 2.3, 3.4.2

[92] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers. *CoRR*, abs/2009.05617, 2020. 1.3, 2.1.2

[93] 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`. 1.3

[94] 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

[95] Lars van Hijfte and Ana Oprescu. Mutantbench: an equivalent mutant problem comparison framework. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 7–12, 2021. doi: 10.1109/ICSTW52544.2021.00015. 6.3

[96] 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. 4, 4.1.2

[97] Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. Can large language models write good property-based tests?, 2023. 1.2, 7

[98] 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, 2.1.2

[99] 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

[100] 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, 5.2.3

[101] 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, 2.1.2, 3.5.1

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

[103] Robert White and Jens Krinke. Reassert: Deep learning for assert generation. *CoRR*, abs/2011.09784, 2020. 2.1.2

[104] 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

[105] 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

[106] Michal Zalewski. american fuzzy lop (2.35b). `http://lcamtuf.coredump.cx/afl/`. Accessed December 20, 2016. 1

[107] 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, 1.3, 2.2, 4, 4.1.1, 4.2, 4.2.1