CAT-LM Training Language Models on Aligned Code And Tests

Nikitha Rao*, Kush Jain*, Uri Alon, Claire Le Goues, Vincent J. Hellendoorn
Carnegie Mellon University
United States
{nikitharao, kdjain, urialon, legoues, vhellendoorn}@cmu.edu

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00193

***Abstract*—Testing is an integral but often neglected part of the software development process. Classical test generation tools such as EvoSuite generate behavioral test suites by optimizing for coverage, but tend to produce tests that are hard to understand. Language models trained on code can generate code that is highly similar to that written by humans, but current models are trained to generate each file separately, as is standard practice in natural language processing, and thus fail to consider the code-under-test context when producing a test file. In this work, 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 analyze its usefulness for realistic applications, showing that sampling with filtering (e.g., by compilability, coverage) allows it to efficiently produce tests that achieve coverage similar to ones written by developers while resembling their writing style. By utilizing the code context, CAT-LM generates more valid tests than even much larger language models trained with more data (CodeGen 16B and StarCoder) and substantially outperforms a recent test-specific model (TeCo) at test completion. Overall, our work highlights the importance of incorporating software-specific insights when training language models for code and paves the way to more powerful automated test generation.**

***Index Terms*—test generation, test completion, large language models, code-test alignment**

I. Introduction

Software testing is a critical component of the software development process. In well-tested projects, most code files are paired with at least one corresponding test file that implements unit and/or integration test functions that evaluate the functionality of the code. However, writing high quality tests can be time-consuming [1], [2] and is often either partially or entirely neglected. This has led to extensive work in automated test generation, including both classical [3], [4], [5], [6] and neural-based methods [3], [7], [8].

Classical test generation tools like EvoSuite [4] directly optimize to generate high-coverage tests. However, the generated tests are often hard to read and may be unrealistic or even wrong [9]. This requires time and effort from developers to verify generated test correctness [5].

Meanwhile, Large Language Models (LLMs) trained on code have made major strides in generating human-like, high-quality functions based on their file-level context [10], [11], [12], [13]. Tools like Copilot excel at code generation, and can significantly improve the productivity of its users [14]. Currently, these models are less well-suited for test generation, because they tend to be trained to generate the code in each file separately, standard practice in natural language processing. Generating meaningful tests, of course, critically requires considering the alignment between the tests and the corresponding code under test. Some prior work on neural-based test generation methods has focused on modeling this alignment [3], [7], [15]. However, this work typically focuses on the relatively narrow task of generating individual assertions in otherwise complete tests, based on a single method under test. Unlocking the more impactful ability to generate entire tests requires leveraging both the entire code file and existing tests as context, which in turn requires substantially larger models.

In this work, we make a significant step towards accurate whole-test generation via CAT-LM, a language model trained on aligned **C**ode **A**nd **T**ests. CAT-LM is a bi-lingual GPT-style LLM with 2.7B parameters. It is trained on a large corpus of Python and Java projects using a novel pretraining signal that explicitly considers the mapping between code and test files, when available, while also leveraging the (much larger) volume of untested code. Modeling the code file along with the test leads to additional challenges regarding a model's context length. Most code generation models support a context window of up to 2,048 tokens. However, our data analysis indicates that many code-test file *pairs* comprise more than 8K tokens. We thus increase the maximum sequence input length, training CAT-LM with a context window of 8,192 tokens.

Our results show that the model effectively leverages the code file context to generate more syntactically valid tests that achieve higher coverage. The model provides a strong prior for generating plausible tests: combined with basic filters for compilability and coverage, CAT-LM frequently generates tests with coverage close to those written by human developers.

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 [16] and, the CodeGen [11] model

*Equal contribution

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00193

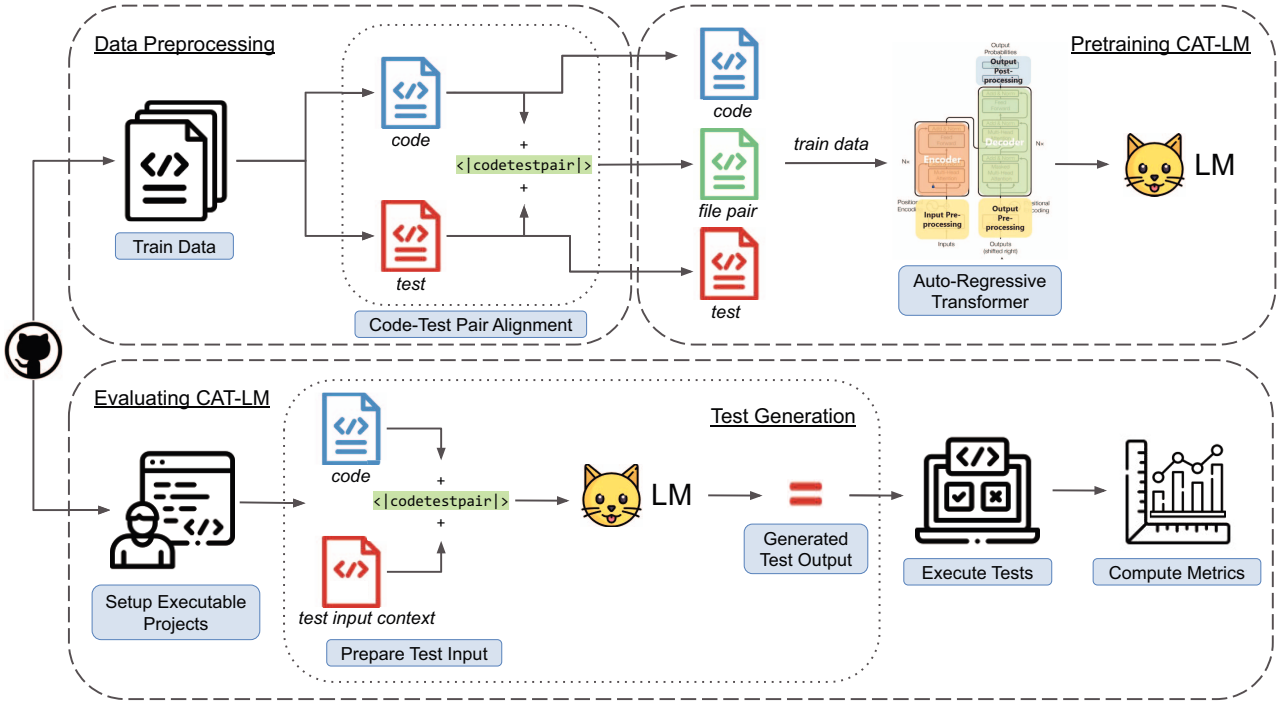


Fig. 1: Approach overview. We extract Java and Python projects with tests from GitHub and heuristically align code and test files (top), which, along with unaligned files, train CAT-LM, a large, auto-regressive language model. We evaluate CAT-LM's generated tests on a suite of executable projects (bottom), measuring its ability to generate syntactically valid tests that yield coverage comparable to those written by developers.

family, which includes mono-lingual models trained on a much larger budget than ours. We also compare against TeCo [15], 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 results highlight the merit of combining the power of large neural methods with a pretraining signal based on software engineering expertise—in this case, the importance of the relation between code and test files.

In summary, we make the following contributions:

- We release a corpus of 1.1M code-test file pairs along with 14.4M Java and Python files across 196K open-source projects. We believe this corpus will be useful for many testing-related tasks.
- We release CAT-LM, the first pretrained LLM that models aligned code and test files from Java and Python projects on GitHub.
- We release the testing framework used to evaluate the tests generated by CAT-LM.
- We conduct an extensive evaluation of CAT-LM with strong baselines on downstream tasks such as test method generation and test completion.

The model checkpoints along with usage code samples can be found at https://github.com/RaoNikitha/CAT-LM.

II. Overview

CAT-LM is a GPT-style model that can generate tests given code context. Figure 1 shows an overview of our entire system, which includes data collection and preprocessing (detailed in Section IV-A), pretraining CAT-LM (Section V), and evaluation (Section VI).

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 IV-A). We filter our training set following CodeParrot [17] standards (including deduplication), resulting in ∼15M code and test files. We align code and test files using a fuzzy string match heuristic (Section IV-B).

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 [18]. We then determine the appropriate model size, 2.7B parameters based on our training budget and the Chinchilla scaling laws [19]. We use the GPT-NeoX toolkit [20] enhanced with Flash Attention [21] 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.

Test generation with code context

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

Fig. 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.

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.

III. Tasks

We describe two tasks for which CAT-LM can be used, namely test method generation (with three settings) and test completion. Figure 2 demonstrates the setup for all tasks including code context.

A. Test Method Generation

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

TABLE I: 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

B. 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).

IV. Dataset

This section describes dataset preparation for both training and evaluating CAT-LM. Table I provides high-level statistics pertaining to data collection and filtering.

A. Data Collection

We use the GitHub API [22] to mine Python and Java repositories that have at least 10 stars and have new commits after January 1st, 2020. Following [23] and [24], 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.

B. 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 [17] 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 [25] between code and test file names, and group them as a pair if they achieve a similarity score greater than 0.85. If multiple matches are found, we keep the pair with the highest score.

Following file pair extraction, we prepare our training data by replacing the code and test files with a new file that concatenates the contents of the code file and the test file, separating them with a unique `<|codetestpair|>` token. This ensures that the model learns the mapping between code and test files from the pretraining signal. Note that we always combine these files starting with the code, so the model (which operates left-to-right) only benefits from this pairing information when generating the test. We additionally include all the other code and test files for which we did not find pairs in our training data, which results in 4.7M Python files and 9.7 Java files. We include these unmatched files to maximize the amount of data the model can learn from. Figure 3 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 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 [26], [27]. 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.

C. 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.

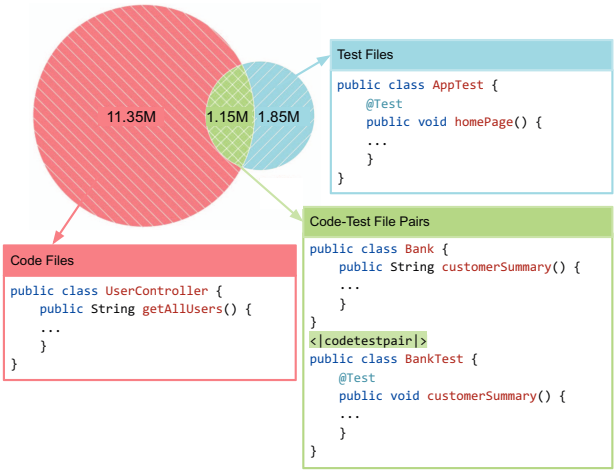


Fig. 3: Distribution of files with sample code snippets

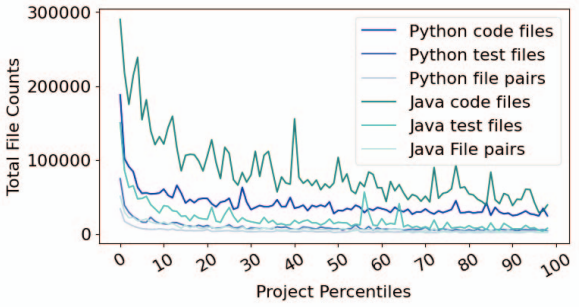


Fig. 4: Distribution of files in projects sorted by GitHub stars, normalized by percentiles

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

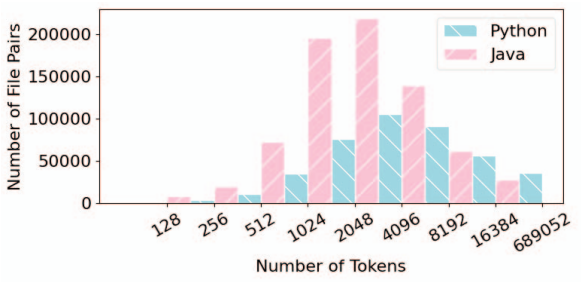


Fig. 5: Distribution of file pair tokens

Python projects. Note that we reuse these Docker containers in our testing framework (See Section VI-A5).

V. CAT-LM

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

A. *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 IV-A). We first train a subword tokenizer [28] using the SentencePiece [18] 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. **Analysing 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 [11], [29].* Our analysis on the distribution of tokens, visualized in Figure 5, showed that this only covers 35% of the total number of file pairs. As such, while it may be appropriate for a (slight) majority of individual files, it would not allow our model to leverage the code file's context while predicting text in the test file. This is a significant limitation since we want to train the model to use the context from the code file when generating tests.

Further analysis showed that approximately 82% of all file pairs for Java and Python have fewer than 8,192 tokens. Since the cost of the attention operation increases quadratically with the context length, we choose this cutoff to balance training cost and benefit. Therefore, we chose to train a model with a longer context window of 8192 tokens to accommodate an additional ∼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.

B. *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 [19], 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 [29], [11], 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 [20] 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 [11], [30];* 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.

C. *Prompting* CAT-LM *to generate outputs:*

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

VI. Experimental Setup

We describe the setup for evaluating CAT-LM across both tasks outlined in Section III, namely test method generation, and test completion.

A. *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 III). We evaluate CAT-LM on test method generation both with code context and, as an ablation, without code context.

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

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

1) Baseline Models: CodeGen is a family of Transformer-based LLMs trained auto-regressively (left-to-right) [11]. Pre-trained 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 [10] on a Python benchmark [11].

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 [16], 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.

2) 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 [31] (longest overlapping subsequence of tokens) and CodeBLEU [32] 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 [15], [33], [34], [35].

3) 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 [3], [7], [15]. See Section 2.2 in supplementary material for additional detailed descriptions of all lexical and run-time metrics.

4) 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.

TABLE II: 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

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. See Section 1 for in supplementary material for coverage distribution of human-written tests.

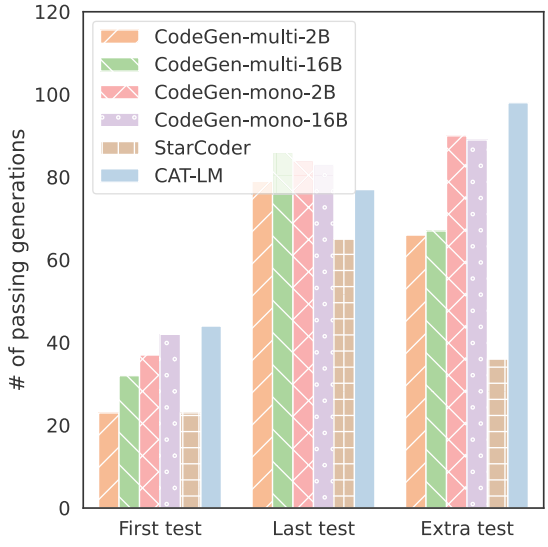
5) Testing Framework: We evaluate the quality of the generated tests using the containers that we setup to execute projects in Section IV-C. We insert the generated test into the original test file, execute the respective project's setup commands and check for errors, recording the number of generated tests that compile and pass the test suite (see Section VI-A3). 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.

B. Test Completion

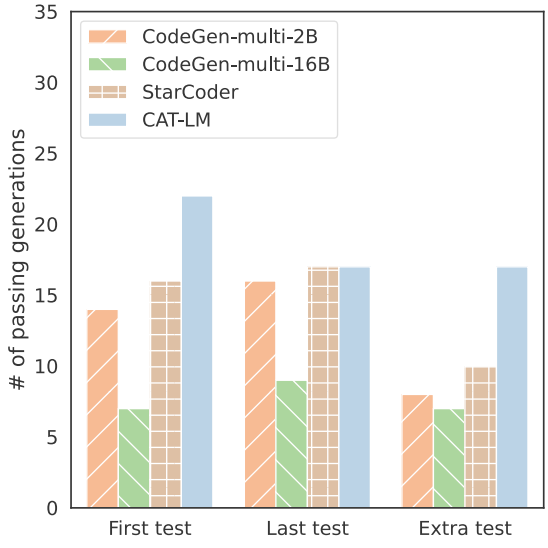
Recall the test completion task involves generating a single line in a given test method, given the test's previous lines. We perform our evaluation for test completion under two conditions, with code context and without code context.

1) Baseline Model: We compare against TeCo [15], a state of the art baseline on test statement completion that has outperformed many existing models, including CodeT5 [35], CodeGPT [36] and TOGA [3]. TeCo [15] is a encoder-decoder transformer model based on the CodeT5 architecture [35]. 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.



(a) Python.



(b) Java.

Fig. 6: Passing tests by model for Python and Java.

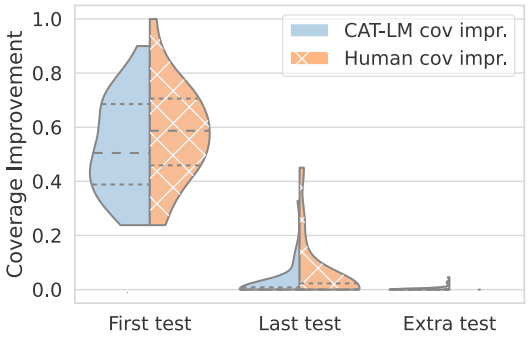
2) Metrics: We compare CAT-LM against TeCo across all lexical metrics (outlined in Section VI-A2).

VII. Evaluation

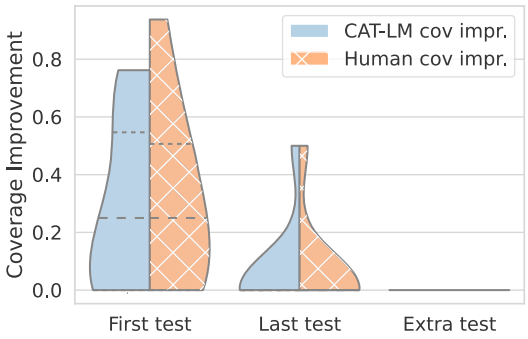
We evaluate CAT-LM's ability to generate valid tests that achieve coverage, comparing against state of the art baselines for both code generation and test completion. Additional results can be found in the supplementary material.

A. Test Method Generation

1) Pass Rate: Figure 6 shows the number of passing tests generated by each model for Python and Java. Note that these



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



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

Fig. 7: Coverage improvement of our model vs humans for different languages.

are absolute numbers, out of a different total for each setting.*

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

*The denominator for each group is the number of file pairs shown in Table II multiplied by 10, the number of samples per context.

use $8,192$ tokens), highlighting that training with code context is important.

2) Coverage: Figure 7 shows the coverage distribution of CAT-LM, contrasted with that of the human-written tests. For both the first test and last test settings, our model performs mostly comparably to humans, with both distributions having approximately the same median and quartile ranges. The extra test task is clearly especially hard: while our model was able to generate many tests in this setting (Figure 6), these rarely translate into *additional* coverage, beyond what is provided by the rest of the test suite, in part because most of the developer-written test suites in our dataset already have high code coverage (average coverage of 78.6% for Java and 81.6% for Python), and may have no need for additional tests. Table II 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.

3) Lexical Similarity: Table III shows 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.

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

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

5) Other Runtime Metrics: Table III also shows 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.

CAT-LM outperforms StarCoder and CodeGen for both Python and Java, **generating more passing tests** on average across all settings. We find that **code context improves performance** across most settings in terms of both lexical and runtime metrics.

B. Test Completion

For test completion (see Section III-B for task definition), we compare CAT-LM against TeCo [15] on the lexical metrics outlined in Section VI-A2. 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 IV 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.

CAT-LM outperforms TeCo across all lexical metrics, with a 40.4% improvement in CodeBLEU score and 36.6% improvement in exact match accuracy. We find that **context only slightly helps** with test statement prediction, indicating that test completion can largely be done without the code under test, in contrast to entire test generation.

See Section 3 and 4 in the supplementary material for additional results on all tasks.

TABLE III: Lexical and runtime metrics performance comparison of the models on the held-out test set for Java and Python. CodeGen refers to CodeGen-multi for Java and CodeGen-mono for Python results. We only report lexical metrics for our first test and last test settings, as there is no gold test to compare against in our extra test setting.

	Java					**Python**				
	Lexical Metrics			**Runtime Metrics**		**Lexical Metrics**			**Runtime Metrics**	
Model	**CodeBLEU**	**XMatch**	**Rouge**	**Compile**	**Pass**	**CodeBLEU**	**XMatch**	**Rouge**	**Compile**	**Pass**
First Test (Total: Java = 270, Python = 1120)										
CAT-LM w Context	41.4%	**15.4%**	60.9%	**50**	**22**	21.0%	0.3%	**39.4%**	**384**	**44**
CAT-LM w/o Context	37.5%	**15.4%**	56.5%	9	9	17.7%	0.4%	30.2%	236	31
Codegen-2B	35.5%	7.7%	56.8%	24	14	18.2%	0.0%	30.9%	259	37
Codegen-16B	42.2%	7.7%	61.8%	25	7	20.8%	0.3%	35.1%	361	42
StarCoder	**44.6%**	10.9%	**62.2%**	28	16	**24.0%**	**1.8%**	38.8%	269	23
Last Test (Total: Java = 180, Python = 930)										
CAT-LM w Context	55.4%	20.8%	70.8%	**54**	**17**	**38.3%**	**4.8%**	**54.9%**	335	77
CAT-LM w/o Context	53.6%	20.8%	68.9%	33	14	33.2%	1.4%	51.9%	**350**	79
Codegen-2B	51.7%	13.0%	69.2%	43	16	36.3%	2.2%	53.2%	326	**84**
Codegen-16B	56.5%	14.3%	**70.9%**	24	9	37.9%	3.4%	54.0%	349	83
StarCoder	**56.9%**	**21.0%**	69.9%	34	**17**	37.6%	4.2%	54.5%	227	65
Extra Test (Total: Java = 270, Python = 1230)										
CAT-LM w Context	–	–	–	**41**	17	–	–	–	380	98
CAT-LM w/o Context	–	–	–	29	**20**	–	–	–	**425**	**104**
Codegen-2B	–	–	–	17	8	–	–	–	376	90
Codegen-16B	–	–	–	15	7	–	–	–	384	89
StarCoder	–	–	–	17	10	–	–	–	269	36

TABLE IV: 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%

C. Qualitative Comparisons

Finally, we conduct a small-scale qualitative case-study of tests generated by CAT-LM, CodeGen-2B-multi [11], GPT-4 [37] and EvoSuite [4]. GPT-4 is a vastly larger language model than ours, trained with an undisclosed budget by OpenAI. EvoSuite is a popular test generation tool for Java based on evolutionary algorithms.

We analyze a randomly sampled passing generation from CAT-LM in contrast to the tests generated by the other tools in the same context across each our three settings (first test, last test and extra test). The tests here are generated for a `Bank` class, which includes methods to add a customer, open an account and print a summary of all accounts and customers. Our goal is to better understand the benefits and drawbacks of each tool's generated tests. Specifically, we look for characteristics of high quality tests, such as meaningful method and variable names, proper invocation of the method under test and high quality assertions. We mainly discuss the first generated test here; for our full set of examples, see Supplemental Materials.

CAT-LM: Listing 1 shows the first test generation by CAT-LM. The name of the test is informative, along with its variables. It also follows unit testing conventions of testing one specific method in the `Bank` class. This is consistent across the examples for last test and extra test. However, for our extra test example, CAT-LM copied the previous test and changed the name of the test method, not testing new functionality.

CodeGen: in Listing 2, the test generated by CodeGen is quite readable, semantically correct, and natural looking. However, it uses multiple non-existent methods from the code under test—a phenomenon popularly dubbed "hallucinating"—since it lacks awareness of `Bank`'s implementation. StarCoder performs similarly, generating tests that are readable, semantically correct, and natural looking but suffer from hallucinations.

GPT-4: GPT-4 consistently performs the best of all three tools, generating tests that either are identical to the ground truth or test new functionality that none of the existing tests do. Listing 3 shows GPT-4's generation for the first test case. Similar to CAT-LM, the GPT-4 generated test has meaningful identifier names and assertions. GPT-4 had similarly good tests for our last test and extra test settings. However, these results come with several caveats. First, GPT-4 was trained on a very large volume of data, including public code, so it is quite likely that it was trained on our test data and has thus seen the original tests.* Second, GPT-4 is a much larger, model, with a training budget orders of magnitude higher than ours. Given our strong performance compared to the (already much more expensive) CodeGen models, we expect that modestly scaling up our training approach could well yield similar or better results.

EvoSuite: EvoSuite performs the worst in all three settings.

*In fact, a similar caveat applies to CodeGen, which we do outperform.

```java
@Test
public void testCustomerSummary() {
    Bank bank = new Bank();
    assertEquals("Customer Summary", "Customer
        Summary", bank.customerSummary());
}
```

Listing 1: CAT-LM first test generation

```java
@Test
public void testBank() {
    Bank bank = new Bank();
    assertEquals(0, bank.getBalance(),
        DOUBLE_DELTA);
    bank.deposit(100);
    assertEquals(100, bank.getBalance(),
        DOUBLE_DELTA);
    bank.withdraw(50);
    assertEquals(50, bank.getBalance(),
        DOUBLE_DELTA);
}
```

Listing 2: CodeGen first test generation

```java
@Before
public void setUp() {
    bank = new Bank();
}

@Test
public void testAddCustomerAndCustomerSummary()
    {
    Customer customer1 = new Customer("John
        Doe");
    bank.addCustomer(customer1);

    String expectedSummary = "Customer
        Summary\n - John Doe (0 accounts)";
    String actualSummary =
        bank.customerSummary();

    assertEquals(expectedSummary,
        actualSummary);
}
```

Listing 3: GPT-4 first test generation

```java
@Test(timeout = 4000)
public void test0() throws Throwable  {
    Bank bank0 = new Bank();
    Customer customer0 = new Customer("v\"PD");
    bank0.addCustomer(customer0);
    Account account0 = new Account(0);
    account0.deposit(148.3628547);
    customer0.openAccount(account0);
    double double0 = bank0.totalInterestPaid();
    assertEquals(0.14836285470000002, double0,
        0.01);
}
```

Listing 4: EvoSuite first test generation

Fig. 8: Example first tests generated by CAT-LM, CodeGen, GPT-4, and EvoSuite. CAT-LM and GPT-4 both generate realistic and readable tests; EvoSuite struggles with poor naming conventions and unrealistic tests. CodeGen generates readable test cases, but hallucinates methods in the code under test. See Section 5 in supplementary material for additional examples.

Listing 4 shows the EvoSuite completion for the bank class. The generated test uses very poor naming conventions, such as naming the method `test0`, and each of the variables `bank0`, `customer0`, and `account0`. The deposit amounts do not make logical sense, as they are not rounded to the nearest cent. There is also a timeout of 4000 milliseconds. Such timeouts are highly likely to lead to flaky tests, where this test might pass in one environment and timeout in a different environment. The other generations by EvoSuite, suffer similar problems, including lacking asserts and using spurious exception handling. Due to this lack of proper naming conventions and the use of trivial asserts, it is very difficult to understand what is being tested in EvoSuite's generation.

Both GPT-4 and CAT-LM generate **high quality tests**, checking for realistic situations with readable asserts. However CAT-LM struggles to generate meaningfully distinct tests in the extra test setting. CodeGen and StarCoder produces highly readable, but incorrect tests. EvoSuite struggles to generate meaningful tests; it uses **poor naming conventions** and **spurious exception handling**.

VIII. Related Work

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 [38], aflplusplus [39] and honggfuzz use coverage to guide generation of test prefixes. Property testing tools such as Korat [40], QuickCheck [41] and Hypothesis [42] allow a developer to specify a set of properties and subsequently generates a suite of tests that test the specified properties. PeX [43] and Eclipser [44] 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 [3], which limits the level of testing they provide. EvoSuite [4] 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 [5], [45], [46].

Neural Test Generation: More recently, neural test generation methods have been developed to generate more natural and human understandable tests. ConTest[8] makes use of a generic transformer model, using the tree representation of code to generate assert statements. ATLAS [7], ReAssert [47], AthenaTest [48] and TOGA [3] 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 [15] 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.

Large Language Models of Code: Large language models (LLMs) can perform well across many tasks when prompted with instructions and examples [49], [30]. Codex [10] is an autoregressive (left to right generation) LLM with 12B parameters, fine-tuned from GPT-3 on 54 million GitHub Python repositories. CodeGen-16B, with which we compare, outperforms this model [11]. Later, unpublished, iterations of Codex have also been applied to commercial settings, powering GitHub's Copilot [14]. TestPilot [50] uses Codex to generate unit tests. However, it requires significant volumes of documentation as input, which is often not available for open-source projects. 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 thus do not learn the alignment of tests to the code under test. We pretrained a comparatively small language model on a much more modest budget that explicitly learns to align code and the corresponding test files, which yields substantially better performance than modestly larger classically trained models.

IX. Conclusion

We develop 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. Despite its strong performance, CAT-LM has limitations including that it may struggle to generalize to unpopular projects, and the comparison with TeCo likely has data leakage (CAT-LM is likely to have seen TeCo's test set in pretraining). We hope that these limitations can be overcome in future work. 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.

Data availability: The model weights for CAT-LM, code and datasets for training and evaluating CAT-LM, results of additional experiments and comparison with TeCo and CodeGen are available at: https://doi.org/10.5281/zenodo.7901830.

X. Acknowledgements

The authors would like to thank Charles Sutton for his mentorship as part of the Google Collab Ph.D. Fellowship, which also included $20,000 in cloud credits without which this work would not have been possible. We additionally thank the authors of TeCo for providing us with data and code for our baseline experiments. This work is supported in part by the US National Science Foundation, awards CCF-2129388 and CCF-1910067.

References

[1] M. Beller, G. Gousios, A. Panichella, and A. 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*, ser. ESEC/FSE '15, 2015, p. 179–190.

[2] M. Beller, G. Gousios, and A. Zaidman, "How (much) do developers test?" in *International Conference on Software Engineering*, ser. ICSE '15, 2015, p. 559–562.

[3] E. Dinella, G. Ryan, T. Mytkowicz, and S. Lahiri, "Toga: A neural method for test oracle generation," in *International Conference on Software Engineering*, ser. ICSE '22, 2022, p. 2130–2141.

[4] G. Fraser and A. 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*, ser. ESEC/FSE '11, 2011, p. 416–419.

[5] C. Brandt and A. Zaidman, "Developer-centric test amplification: The interplay between automatic generation human exploration," *Empirical Software Engineering*, vol. 27, no. 4, 2022.

[6] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Computing Survey*, vol. 51, no. 3, pp. 50–88, 2018.

[7] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," *CoRR*, vol. abs/2002.05800, 2020.

[8] J. Villmow, J. Depoix, and A. Ulges, "ConTest: A Unit Test Completion Benchmark featuring Context," in *Workshop on Natural Language Processing for Programming*, Aug. 2021, pp. 17–25.

[9] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities," in *International Conference on Software Maintenance and Evolution*, 2020, pp. 523–533.

[10] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. W. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, I. Babuschkin, S. A. Balaji, S. Jain, A. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," *CoRR*, vol. abs/2107.03374, 2021.

[11] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A Conversational Paradigm for Program Synthesis," *CoRR*, vol. abs/2203.13474, 2022.

[12] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "InCoder: A Generative Model for Code Infilling and Synthesis," *CoRR*, vol. abs/2204.05999, 2022.

[13] M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. McLeavey, J. Tworek, and M. Chen, "Efficient Training of Language Models to Fill in the Middle," *CoRR*, vol. abs/2207.14255, 2022.

[14] "GitHub Copilot," 2021. [Online]. Available: https://github.com/features/copilot

[15] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, "Learning deep semantics for test completion," in *International Conference on Software Engineering*, ser. ICSE '23, 2023, p. 2111–2123.

[16] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J.

Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023.
[17] L. v. Werra, "Codeparrot," https://github.com/huggingface/transformers/tree/main/examples/research_projects/codeparrot.
[18] "SentencePiece." [Online]. Available: https://github.com/google/sentencepiece
[19] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark *et al.*, "Training compute-optimal large language models," *CoRR*, vol. arXiv:2203.15556, 2022.
[20] "GPT-neox Toolkit." [Online]. Available: https://github.com/EleutherAI/gpt-neox
[21] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Advances in Neural Information Processing Systems*, 2022.
[22] "GitHub REST API." [Online]. Available: https://docs.github.com/en/rest
[23] M. 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*, ser. SPLASH '19, 2019, pp. 143–153.
[24] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: a map of code duplicates on GitHub," in *Proceedings of the ACM on Programming Languages*, ser. OOPSLA '17, vol. 1, 2017, pp. 1–28.
[25] "TheFuzz: Fuzzy String Matching in Python." [Online]. Available: https://github.com/seatgeek/thefuzz
[26] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *International Conference on Quality Software*, ser. ICQS '13, 2013, pp. 103–112.
[27] H. H. F. d. Souza, I. Wiese, I. Steinmacher, and R. Ré, "A characterization study of testing contributors and their contributions in open source projects," in *Brazilian Symposium on Software Engineering*, ser. SBES '22, 2022, pp. 95–105.
[28] T. Kudo, "Subword regularization: Improving neural network translation models with multiple subword candidates," in *Annual Meeting of the Association for Computational Linguistics*, ser. ACL '18, 2018, pp. 66–75.
[29] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A Systematic Evaluation of Large Language Models of Code," *CoRR*, vol. abs/2202.13169, 2022.
[30] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and Efficient Foundation Language Models," *CoRR*, vol. abs/2302.13971, 2023.
[31] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Conference on Text Summarization Branches Out*, 2004, pp. 74–81.
[32] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "CodeBLEU: a Method for Automatic Evaluation of Code Synthesis," *CoRR*, vol. abs/2009.10297, 2020.
[33] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with Hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, 2020.
[34] A. LeClair, S. Jiang, and C. McMillan, "A Neural model for generating natural language summaries of program subroutines," in *International Conference on Software Engineering*, ser. ICSE '19, 2019, pp. 795–806.
[35] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.
[36] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.
[37] OpenAI, "Gpt-4 technical report," 2023.
[38] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Conference on Object-Oriented Programming Systems and Applications Companion*, ser. OOPSLA '07, 2007, pp. 815–816.
[39] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Conference on Offensive Technologies*, ser. WOOT '20, 2020, pp. 10–10.
[40] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," *SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 123–133, 2002.
[41] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *International Conference on Functional Programming*, ser. ICFP '00, 2000, p. 268–279.
[42] D. MacIver, Z. Hatfield-Dodds, and M. Contributors, "Hypothesis: A New Approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 2019.
[43] N. Tillmann and P. de Halleux, "Pex - white box test generation for .net," in *Tests and Proofs*, ser. TAP '08, vol. 4966, April 2008, pp. 134–153.
[44] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *International Conference on Software Engineering*, ser. ICSE '19, 2019, pp. 736–747.
[45] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children Thing1 and Thing2?" in *International Symposium on Software Testing and Analysis*, ser. ISSTA '17, 2017, pp. 57–67.
[46] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. 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*, ser. ASE '11, 2011, pp. 23–32.
[47] R. White and J. Krinke, "Reassert: Deep learning for assert generation," *CoRR*, vol. abs/2011.09784, 2020.
[48] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers," *CoRR*, vol. abs/2009.05617, 2020.
[49] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are Few-Shot learners," in *Advances in Neural Information Processing Systems*, 2020, pp. 1877–1901.
[50] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "Adaptive Test Generation Using a Large Language Model," *CoRR*, vol. abs/2302.06527, 2023.