Software Lab
Institute of Software Engineering
University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart

Master Thesis

# Tests4J Benchmark: Execution-based Evaluation of Context-Aware Language Models for Test Case Generation

Valentin Knappich

| | |
|---|---|
| **Course of study:** | Computer Science |
| **Examiner:** | Prof. Dr. Michael Pradel |
| **Supervisor:** | Prof. Dr. Michael Pradel, Matteo Paltenghi |
| **Started:** | October 17, 2022 |
| **Completed:** | April 17, 2023 |

University of Stuttgart
Germany

SOLA
SoftwareLab

# Abstract

Testing is a critical part of the software engineering process. The cost associated with writing and maintaining test suites has motivated the research community to develop approaches to automatically generate test cases for a given software. Evosuite is one of the most established tools for Java and has been shown to achieve high coverage. However, the test cases lack readability, motivating the application of language models (LMs) of code to the task. Evaluating such neural test case generators based on their execution requires substantial efforts to set up evaluation projects and to obtain metrics like coverage from them. In consequence, most prior work on Java test case generation has either evaluated models on a small number of selected methods under test (MUTs) or used Defects4J as an evaluation benchmark. However, small benchmarks suffer from high variance, and many projects in Defects4J have been used to pre-train LMs of code. To fill that gap, we introduce Tests4J, a novel benchmark for neural and non-neural test case generators. Tests4J contains 12k test cases from 60 Java projects, out of which 41 are used for training while 19 are used for evaluation. For all projects, it includes the complete repository, enabling execution-based evaluation and open-ended experimentation with project-specific context information. In a single command, Tests4J allows researchers to obtain execution-based metrics like coverage and intrinsic metrics like loss, BLEU and crystalBLEU.

Using Tests4J, we train and evaluate several test case generation models based on PolyCoder with 400M parameters. We compare Evosuite to our best neural model and find that the individual test cases achieve similar coverage. However, Evosuite generates 3 times as many test cases, covering about 3 times as many lines in total. We furthermore find that Evosuite fails to generate any test cases for 4 out of 11 projects in the test set. This presents a fundamental advantage of LMs: they do not need to integrate with the project and thus don't suffer from dependency conflicts. Next, we evaluate prefix tuning as a training method and find that there is a significant gap to full fine-tuning. We further investigate the importance of project-specific context information and create simplified representations of the focal class and the test class. We find that adding this context information increases the achieved coverage by more than 4x, and that the focal class and test class context are highly complementary. Motivated by this finding, as well as the hard token limit and quadratic complexity of transformers, we propose Context-Aware Prompt Tuning (CAPT). In CAPT, context information is first compressed into embeddings, and then injected into the LM as soft tokens, similar to prefix tuning. We find that the method does not yield significant improvements over the baseline, but present directions for future research. Lastly, we find that loss is not an ideal indicator of coverage and that there is a high variance in coverage between projects, and thus advocate for large-scale execution-based evaluations.

# Zusammenfassung

Softwaretests spielen eine zentrale Rolle in der Softwareentwicklung. Neben etablierten Tools wie Evosuite wurden in den letzten Jahren vermehrt Sprachmodelle für die automatische Generierung von Testfällen eingesetzt. Die ausführungsbasierte Evaluierung solcher Modelle stellt einen erheblichen Aufwand dar. Infolgedessen haben vorherige Arbeiten Modelle entweder auf einer kleinen Auswahl zu testender Methoden oder auf Defects4J evaluiert. Wir halten beide Ansätze für suboptimal, da Evaluierungen auf kleinen Datensätzen eine hohe Varianz aufweisen und die Projekte in Defects4J zu einem großen Anteil in weit verbreiteten Datensätzen für das Vor-Trainieren von Sprachmodellen enthalten sind. Um diese Defizite zu beheben, führen wir in dieser Arbeit Tests4J ein. Es handelt sich dabei um einen neuen Benchmark zur Evaluierung von neuronalen und nichtneuronalen Methoden für die Generierung von Tests. Tests4J beinhaltet 12k Testfälle aus 60 Java Projekten, wobei 41 für das Training und 19 für die Evaluierung eingesetzt werden. Tests4J enthält eine komplette Kopie aller Projekte, sodass eine ausführungsbasierte Evaluierung und umfassende Experimente mit Kontextinformationen ermöglicht werden. In einem einzigen Befehl können mit Tests4J sowohl ausführungsbasierte Metriken wie die Abdeckung als auch intrinsische Metriken wie Loss, BLEU und crystalBLEU berechnet werden.

Mithilfe von Tests4J trainieren und evaluieren wir verschiedene Modelle auf Basis von PolyCoder mit 400M Parametern. Wir vergleichen Evosuite mit unserem besten neuronalen Modell und stellen fest, dass einzelne Testfälle ähnlich effektiv in der Abdeckung sind. Jedoch generiert Evosuite dreimal so viele Testfälle und erreicht die dreifache Abdeckung insgesamt. Zudem evaluieren wir Prefix Tuning als Trainingsmethode und stellen einen signifikanten Unterschied zum Trainieren des ganzen Modells in Bezug auf die Abdeckung und die intrinsischen Metriken fest. Wir untersuchen zudem die Bedeutung von projektspezifischen Kontextinformationen und erstellen vereinfachte Darstellungen der zu testenden Klassen und Testklassen. Dabei kommen wir zu dem Ergebnis, dass Kontextinformationen essenziell sind und zu mehr als der vierfachen Abdeckung führen. Motiviert durch diese Erkenntnis sowie durch die limitierte Anzahl an Tokens und die quadratische Komplexität von Transformern, schlagen wir Context-Aware Prompt Tuning vor. Dabei wird der Kontext zunächst von einem kleineren Modell in numerische Repräsentationen komprimiert, welche dann als virtuelle Tokens vom Sprachmodell verarbeitet werden. Wir stellen fest, dass die Methode keine signifikanten Verbesserungen gegenüber der Baseline erzielt, zeigen aber Richtungen für zukünftige Forschung auf. Abschließend stellen wir fest, dass Loss kein idealer Indikator für den Abdeckungsgrad ist, sowie dass es eine große Varianz im Abdeckungsgrad zwischen Projekten gibt und plädieren daher für umfassende, ausführungsbasierte Evaluierungen.

# Contents

# Acronyms

**AST** Abstract Syntax Tree. 17

**CAPT** Context-Aware Prompt Tuning. 3, 24

**LM** Languge Model. 1, 3, 5–7, 9, 11, 15, 24, 49

**MLP** Multilayer Perceptron. 7

**MUT** method under test. 1, 8–10, 12, 14, 17, 18, 20, 23

**NLP** Natural Languge Processing. 1, 6

**PEFT** parameter-efficient fine-tuning. 7

**PG** Prompt Generator. 24

**SGD** Stochastic Gradient Descent. 1, 7

# 1 Introduction

Testing is a critical part of the software engineering process. Unit tests are at the bottom of the testing pyramid [13] and test single functions or methods called methods under test (MUTs). In modern codebases, test code constitutes a significant amount of code, often spanning more lines than the main code. The cost associated with writing, updating and maintaining this test code motivated the research community to develop tools for automatic test case generation. These tools automatically generate test cases for MUTs based on their signature, body, and sometimes additional context information from the class or project. Evosuite [19] and Randoop [36] are the most popular tools for Java. Evosuite uses evolutionary algorithms to generate test suites, whereas Randoop uses a random search directed by feedback from executing the tests. Typically, test suites generated by these tools achieve excellent coverage scores, sometimes even outperforming human developers in that regard [20]. However, there are some limitations that hinder the adoption of these tools in practice. One major concern is the quality of the generated test cases. Analyses revealed multiple test smells that appear significantly more often in generated tests than in manually written ones [22, 38]. Generated test cases have been shown to be less readable and understandable [23, 14], arguably increasing the amount of work necessary for maintaining test suites. In a survey, most industrial practitioners said they would not keep a generated test case without modification, citing readability as one of the main reasons [4].

Meanwhile, the Natural Languge Processing (NLP) research community developed powerful Language Models (LMs) able to perform both discriminative and generative tasks on natural language. These models, usually based on the Transformer architecture [48], are pre-trained on self-supervised objectives and later adapted to downstream tasks such as text summarization. For such adaptation, there are two main approaches: fine-tuning and in-context learning. In fine-tuning, the LM weights are further trained on a downstream task objective, whereas in-context learning [32] achieves adaptation merely by providing a task description or input-output examples in the input, also called prompt. Soft prompt tuning is a family of methods that aim to find a middle ground between these two directions by introducing additional parameters in the form of soft tokens. Thereby, soft tokens refer to continuous embeddings that are treated as tokens but trained directly with Stochastic Gradient Descent (SGD). Accordingly, normal textual prompts are called hard prompts.

The success of LMs in the field of NLP has led to the adoption in the domain of source code as well. The result are LMs of code that can reason about and generate source code [18, 35, 50, 12, 52, 5]. They have been applied to many tasks, code completion, code synthesis and bug detection being among the most popular ones [40]. Yet, only relatively few works have attempted to generate

```java
public T retain() throws RefCountResourceException {
    synchronized (this) {
        if (refCount.getAndIncrement() == 0) {
            try {
                log.debug("First Reference. Create Resource.");
                resource.set(resourceSupplier.get());
            }
            catch (Throwable throwable) {
                throw new RefCountResourceException(throwable);
            }
        }

        return resource.get();
    }
}

@Test
public void retainShouldNotCreateResourceOnSecondCall() throws Throwable {
    AtomicInteger refCount = new AtomicInteger(1);
    RefCountResource resource = new RefCountResource(resourceSupplier, resourceCleanup, refCount,
    ↪  new AtomicReference(new Object()));
    resource.retain();

    verify(resourceSupplier, times(0)).get();
    assertEquals(2, refCount.get());
}
```

Figure 1.1: Example MUT and test case that requires substantial context information.

test cases with LMs [6, 47, 2, 43]. In these works, there is no clear consensus on the evaluation methodology. Some works evaluate models using intrinsic metrics like the loss [47, 2]. Such metrics, loss in particular, are easy to compute, enabling simple evaluations on a large scale. However, we argue that ultimately, a test case should cover the MUT and correctly assert its behavior, which can be achieved in many ways. Since intrinsic metrics only compare a generated test case to a ground truth, there is the risk of a disconnect between these metrics and the final performance. Execution-based metrics offer a more reliable evaluation because they are not restricted to a single ground truth and directly measure coverage as one key quality attribute of a test case. However, performing coverage analyses is associated with substantial effort to set up. Prior work has therefore either focussed on coverage evaluation on a small number of classes and projects [6], or used existing benchmarks [47, 2] like Defects4J [27]. We argue that small-scale evaluation setups suffer from high variance in coverage between classes and even between projects. Defects4J, on the other hand, contains a large portion of repositories that have been used to pre-train LMs, potentially leading to data leakage and distorted results. To fill that gap, we introduce Tests4J, a large-scale benchmark for neural and non-neural test case generators that provides execution-based and intrinsic metrics, while avoiding data leakage.

To apply LMs to the task of test case generation, it is cast to a completion task: the language model is given a prompt and asked to complete it. In the most basic setup, this prompt only contains the MUT. We argue that this setup is ill-posed because the MUT is only part of the information required to write correct test cases. Figure 1.1 shows an example where the test case

requires substantial context information beyond the MUT. For instance, it requires information on how to instantiate objects of the focal class `RefCountResource`, as well as information on the initialized mock objects `resourceSupplier` and `resourceCleanup` from the test class. Illustrated by this example and empirically shown by prior work [47], providing such context information is therefore essential for generating test cases. However, transformer LMs are limited in processing context by hard token limits of 1024 or 2048 and the quadratic compute and memory complexity. We attempt to alleviate these issues and introduce Context-Aware Prompt Tuning (CAPT), which first compresses context information into embeddings and injects them into the LM as virtual tokens, similar to soft prompt tuning. We summarize our main contributions to be the following:

1. We introduce Tests4J, containing 12k pairs of MUTs and test cases from 60 Java projects. Out of these, 19 projects are set up to be executable in a fully automatic way, enabling both intrinsic and execution-based evaluation in a single command. The remaining projects are used for training. All projects are stored as complete clones, allowing extensive experimentation with context information.

2. We integrate Evosuite into Tests4J, enabling the comparison of Evosuite and our neural approaches (RQ4). We find that individual test cases generated by the two approaches are similarly effective in covering the MUTs. However, Evosuite generates about 3 times as many executable test cases, covering about 3 times as many lines in total.

3. Based on Tests4J, we experiment with prefix tuning (RQ1) and the importance of context information (RQ2). We find that prefix tuning lacks significantly behind full fine-tuning and that good context representations from both focal and test class are essential.

4. We propose and evaluate CAPT, finding that it is unable to effectively retain performance while reducing the sequence length. We present hypotheses for the reasons and leave further experimentation to future work.

The remainder of this thesis is structured as follows. In chapter 2, we provide background knowledge on test case generation, LMs and Prompt Tuning. Chapter 3 describes the construction of the benchmark, whereas Context-Aware Prompt Tuning is described in chapter 4. The experimental setup, research questions and results are presented in chapter 5. In chapter 6, we discuss the results and derive directions for future work. Chapter 7 describes related work and chapter 8 summarizes the experiments and findings of this work.

# 2 Background

This chapter contains background information on the most relevant topics in this thesis: Language Models, Prompt Tuning and Test Case Generation.

## 2.1 Language Models

Language Models (LMs) are machine learning models trained to predict the probability of a given text. They are usually trained in a self-supervised manner using a denoising objective like masked language modeling or causal language modeling. Generative LMs, which this work focuses on, often use the former. At every time step, the model predicts the most likely next token. To generate text, the prediction is applied autoregressively, i.e., the model generates text token by token from left to right. Generally, the inputs and outputs of LMs are sequences of tokens. We denote such sequences by $x_{i:j}$, which refers to all tokens with indices inclusively between $i$ and $j$: $x_{i:j} = [x_i, ..., x_j]$. In a sequence-to-sequence task, such as test case generation, the model is trained to infer a target sequence $y_{0:m}$ from a prompt $x_{0:n}$. Figure 2.1 illustrates this setup and introduces the model visualization used throughout this thesis. More formally, we define the model to predict a conditional probability distribution $p(y_{i+1}|x_{0:n}, y_{0:i})$. We can then formulate the cross entropy loss as

$$CE = -\sum_{i=0}^{m} log(p(y_{i+1}|x_{0:n}, y_{0:i}))$$

Since the introduction of the transformer architecture [48], it has been the predominant architecture for LMs. It is based on self-attention, where every token is represented by an embedding. At every layer, every token embedding is used to create query, key, and value embeddings via linear projections. For every token, the dot-product similarity of its query embedding with all other key embeddings determines how much each of the other tokens should influence this token embedding, i.e., how much attention it should pay to each other token. The new token embedding is then the weighted sum of each value embedding, weighted by attention score. More formally, the authors formulate the attention mechanism in a compact way by combining the query, key, and value embeddings for every token into the matrices Q, K, and V:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Instead of performing the attention mechanism with single, large query, key, and value embeddings per token, the authors propose multi-head self-attention. There, self-attention as described above

is performed for multiple smaller query, key, and value embeddings that are created by separate projections.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

The self-attention mechanism has one major drawback compared to earlier approaches based on recurrence or convolutions. It has a quadratic complexity in terms of both compute and memory with respect to the sequence length. That is, using longer prompts with more context information comes at the cost of much higher resource requirements. While the notation of $\mathcal{O}(n^2)$ is generally about scaling behavior in the limit of $n$, quadratic behavior can already be clearly observed within the range of $[0, 2048]$ (see e.g. Figure 4.1), motivating our work on CAPT in chapter 4.

The original transformer proposed by Vaswani et al. [48] was an encoder-decoder architecture, i.e., an encoder first processed the prompt and a decoder generated new output while flexibly attending to both prompt and previous outputs. Subsequently, many architectures were derived that use only an encoder (e.g. BERT [15]), only a decoder (e.g. GPT-2 [41]) or both (e.g. T5 [42], BART [30]). Motivated by the success of pre-trained transformer LMs in NLP, they were also adopted for code-related tasks, resulting in LMs of code. In that context, code is, like natural language, represented as a sequence of tokens. Some of the most notable pre-trained models are Codex [12], PLBART [1], CodeGen [35] and PolyCoder [52]. In this work, we focus on the PolyCoder family of models based on the GPT-2 architecture. The authors publicly release three model variants with 160M, 400M and 2.7B parameters. Such models have been applied to several tasks. For instance, code completion poses the task of continuing an incomplete snippet of code, code generation is the task of generating code from a natural language description and code summarization is the task of generating a natural language description for code [40]. LMs of code are usually pre-trained on a corpus of code that is as general and representative as possible. In that sense, such corpora usually contain code in a number of programming languages and from many domains. To specialize models to a specific downstream task, such as Java test case generation, there are two main paradigms of adaption. First, there is fine-tuning, where all or some of the model parameters are adjusted using task-specific losses. Second, there is in-context learning, where the model is merely conditioned on a task via prompt, while the parameters remain unchanged. To perform in-context learning, the prompt might for instance contain a natural language task description or input-output pairs. If there are multiple of these pairs in the prompt, the method is called few-shot prompting. Depending on the task, fine-tuning often leads to equally accurate but much smaller models. On the other hand, fine-tuning requires sufficiently large datasets and results in the deployment of many models if multiple tasks should be supported.

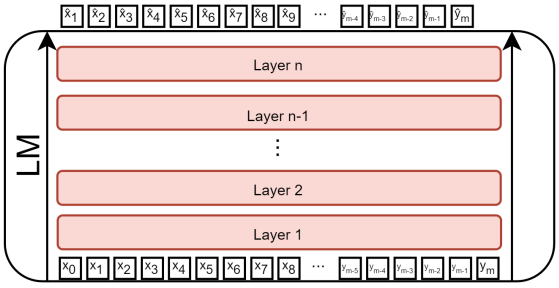



Figure 2.1: Transformer Language Model. $x_{0:n}$ represents the prompt and $y_{0:m}$ the target. Red boxes represent trainable parameters, blue boxes represent frozen parameters, and black boxes represent tokens.

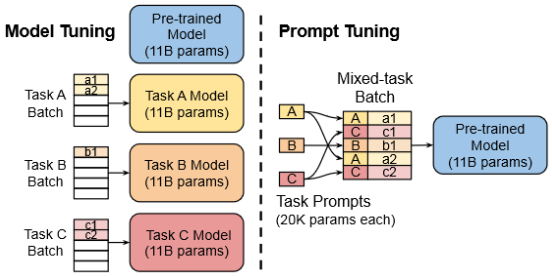



Figure 2.2: Deployment benefits of soft prompt tuning. Figure from [29].

## 2.2 Soft Prompt Tuning

To reduce the complexity of storing and deploying fine-tuned models for multiple tasks, and to reduce computational requirements for fine-tuning, parameter-efficient fine-tuning (PEFT) methods have been proposed. In PEFT, only a small number of parameters are tuned, while the others remain unchanged. The tuned parameters are either newly introduced and randomly initialized or selected among the existing ones [11]. Soft prompt tuning is a family of PEFT methods. Inspired by the success of discrete prompts in models like GPT-3 [10], they add soft tokens to frozen LMs. These soft tokens are continuous embeddings and can therefore be directly optimized using SGD. Whereas only a small portion of the total parameters are fine-tuned, these methods have been shown to achieve comparable performance to full fine-tuning in many tasks [31, 29, 33]. At the same time, they open up interesting opportunities in the model deployment. Practitioners can deploy a single model endpoint of the main model that supports multiple tasks by selecting the respective soft prompts. As depicted in Figure 2.2, one can even mix different tasks in the same batch. Additionally, the memory consumption during fine-tuning is slightly reduced because the optimizer does not need to maintain states for the parameters of the main model (gradients of the main model are still required to allow backpropagation through the model to the soft tokens). The two most prominent methods in this family are Prefix Tuning [31] and Prompt Tuning [29]. They mostly differ in the shape of the soft prompts and how they are injected into the LM, as depicted in Figure 2.3. Prompt Tuning trains soft tokens of the size of the hidden dimensionality and treats them as input embeddings. The hidden states of subsequent layers are determined by the attention mechanism, just as with regular tokens. In contrast, the soft tokens learned by Prefix Tuning span all layers of the LM. Specifically, they contain key and value embeddings for every layer that are directly injected into the attention mechanism. The authors of Prefix Tuning argue that this increases the expressiveness of the method. To stabilize the training process, the prefix embeddings are reparameterized: every token is represented by an embedding of the hidden dimensionality, and projected up to the target size by a Multilayer Perceptron (MLP) that is shared among tokens.

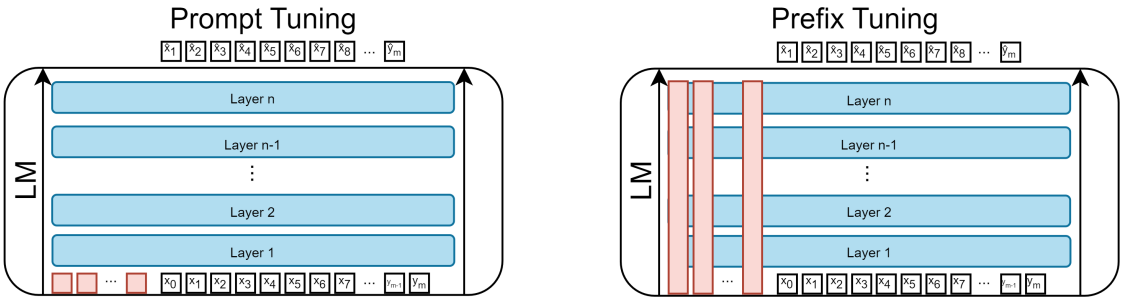



Figure 2.3: Comparison of Prompt Tuning and Prefix Tuning. Prefix Tuning is shown at inference time, i.e., the reparameterization with the MLP is not depicted.

## 2.3 Test Case Generation

Unit test cases are functions that test the functionality of software. Contrary to integration tests and end-to-end tests, unit tests test a small piece of code, usually functions or methods. They generally follow four phases: setup, execution, validation, tear-down [9]. In object-oriented languages such as Java, test cases are often organized in test classes, where some of the setup code is shared among test cases. Besides general software quality factors, the quality of test cases are usually quantified using metrics like coverage and mutation scores. Coverage generally measures what portion of the code under test was executed by a test case or test suite. In particular, it measures the fraction of lines or branches that are executed, yielding the line coverage and branch coverage metrics. Whereas coverage is an important criterion for the quality of test cases, it does not consider the quality of the assertions. In contrast, mutation scores mutate the code under test, check if it still passes the test case and can thereby quantify how likely it is that a regression is detected by the test.

Test case generation refers to the task of automatically generating test cases for a given MUT. In the last decades, several approaches for test case generation have been proposed. For Java, Evosuite [19] and Randoop [37] have been the predominantly used tools. Randoop's approach uses random testing, where the test cases are generated by randomly sampling sequences of method calls. Furthermore, the random generation is guided by feedback from the execution in order to get sequences that are executable and not redundant with respect to the program state. In contrast, Evosuite uses a genetic algorithm that first generates a set of seed tests and generates new tests by applying crossover and mutation operators to the population. The fitness of tests is determined using coverage feedback, i.e., Evosuite directly optimizes for code coverage. Consequently, Evosuite typically achieves very high coverage scores, sometimes even outperforming human developers in that regard [20].

# 3 Tests4J Benchmark

We introduce Tests4J, a benchmark for training and evaluating neural test case generation models. In essence, the benchmark consists of a dataset and an evaluation pipeline. The dataset contains mappings from MUTs to test cases, whereas the evaluation pipeline automatically obtains coverage metrics for generated test cases. We restrict our work to the Java programming language. It is one of the most used languages in prior work regarding test case generation and also provides strong non-neural baselines such as Evosuite [19]. To enable the evaluation pipeline in obtaining execution-based metrics, we include complete repository snapshots in the dataset, rather than just pairs of MUTs and test cases. This also makes the dataset extensible regarding the additional context information a model could use.

Prior work [47, 2] has used a subset of Defects4J [27] to evaluate test case generators with coverage metrics. However, many of the repositories have been used to pre-train LMs of code. For instance, the popular pre-training dataset The Pile [21] contains parts of 16 out of 17 repositories in Defects4J. Similarly, PolyCoder's [52] pre-training corpus contains parts of 12 out of 17 repositories. Evaluating on these repositories is therefore problematic because they might have memorized the test case rather than inferring them. In this benchmark, we remove all repositories used in these two datasets from the candidates. Furthermore, the dataset creation process is mostly automated, enabling future research to increase the scale and avoid data leakage from other pre-training corpuses as well. Furthermore, Defects4J does not contain ground truth test cases. That is, it does not enable evaluation with intrinsic metrics like crystalBLEU [17]. To evaluate a novel test case generator through execution, researchers can insert generated test cases into the respective repositories.

Another popular benchmark for test case generators is JUGE [16]. It is used annually for the JUnit testing tool competition[1]. At its core, JUGE is a standardized infrastructure to measure the effectiveness and efficiency of test case generators. It has been designed for and applied to a variety of methods, inlcuding search-based approaches (e.g. Evosuite [19]) and random-based approaches (e.g. Randoop [36]). However, it has not been used to evaluate neural approaches. We argue that its design is fundamentally not ideal for neural test case generators for multiple reasons. First, much like Defects4J, the majority of the repositories that have been used as benchmark in the last years can be found in the pre-training corpus of many LMs. Second, one key pillar of JUGE is a standardized execution environment that is not sufficient to run LMs. Lastly, JUGE requires developers to write Java interfaces for their test case generators while most deep learning research is done in Python. In contrast, the only interface between models and Tests4J is JSON with a

[1] https://junitcontest.github.io/

| Benchmark | Training Data | Intrinsic Evaluation | Execution-based Evaluation | Accounts for Data Leakage | Interface between Generator and Benchmark |
|---|---|---|---|---|---|
| Methods2Test [46] | ✓ | ✓ | - | - | - |
| Defects4J [27] | - | - | ✓ | - | - |
| JUGE [16] | - | - | ✓ | - | Java |
| Tests4J (ours) | ✓ | ✓ | ✓ | ✓ | JSON |

Table 3.1: Comparison of Test Case Generation Benchmarks and Datasets

simple schema, allowing quick adoption.

Methods2Test [46] is a dataset with pairs of MUTs and test cases intended for training and intrinsic evaluation. We improve upon Methods2Test by including complete real-world projects, enabling execution-based evaluation and open-ended context. In contrast, Methods2Test only contains pairs of MUTs and test cases, and does not provide commit hashes or scraping dates. Therefore, it does not allow execution-based evaluation and provides fixed, limited contextual information. However, we build upon Methods2Test in two main ways. First, we use their list of repositories as a starting point, which assures that all our repositories are using licenses that allow redistribution and that they are not forks. Second, we re-use their heuristic used to map test cases to focal methods based on matching file paths, method names, and method calls.

Table 3.1 summarizes the comparison with the aforementioned existing benchmarks and datasets. The remainder of this chapter describes both the creation of the dataset and the process of obtaining coverage metrics for generated test cases. We start with a set of candidate repositories and filter it according to our requirements, as illustrated in section 3.1. Next, we describe the dataset split in section 3.2 and present the final dataset in section 3.3. Afterward, we elaborate on the process of inserting generated test cases into the repository and getting coverage metrics in section 3.4. Finally, we introduce our evaluation metrics in section 3.5 and present the coverage scores for the ground truth test cases in section 3.6.

## 3.1 Filtering Repositories

The overall goal of the filters is to ensure that all repositories meet the requirements described above. We start with a list of candidates, which contains the 9410 repositories from Methods2Test as well as 9 additional repositories that were used by Bareiß et al. [6] to evaluate their approach to test case generation. The filters are executed in two main steps. First, the repositories are filtered based on their metadata, which we query from the GitHub API[2] (steps 1-5). Afterward, the repositories are cloned and filtered, e.g., based on the number of mappable test cases and whether they are executable (steps 6-9). Table 3.2 summarizes the filtering process. We apply the 9 filters described below, after which 63 out of the 9419 candidate repositories remain. In the following, all quantitative results and plots refer to the state of the dataset at the respective stages in the process, not to the final dataset. Section 3.3 presents further information about the final dataset.

[2] https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28

| | | Filter Name | Before | After | Relative Reduction |
|---|---|---|---|---|---|
| 1 | GitHub API | Data Leakage | 9419 | 6428 | -31.8% |
| 2 | | Existence | 6428 | 6277 | -2.3% |
| 3 | | Repository Size | 6277 | 6163 | -1.8% |
| 4 | | Maven Usage | 6163 | 3272 | -46.9% |
| 5 | | Last Commit Date | 3272 | 1198 | -63.4% |
| 6 | Cloned | Number of Tokens | 1198 | 1193 | -0.4% |
| 7 | | Number of Test Cases | 1193 | 305 | -74.4% |
| 8 | | Compilation | 305 | 152 | -50.2% |
| 9 | | Execution | 152 | 63 | -58.6% |

Table 3.2: Number of Repositories before and after the respective filtering steps

1. **Data Leakage.** To avoid data leakage, we first remove all candidate repositories that appear in the pretraining corpus of two LMs that were considered for this work: Polycoder [52] and GPT-Neo [7]. This eliminates almost a third of the repositories.

2. **Existence.** We ensure that the repository still exists by calling the GitHub API, eliminating about 2.3% of the candidates.

3. **Repository Size.** We find that there is a small number of very large repositories that take a long time to clone and compile, slowing down the process of scraping as well as coverage evaluation. Thus, with a threshold of 300MB, we reduce the total size of candidate repositories by over 60% while reducing the number of candidates by only 1.8%. Figure 3.1 shows a histogram of the repository sizes. Note that the size returned by GitHub's API corresponds to the size that is downloaded during cloning, i.e., a compressed version of the repository, including its history. The history is not as relevant for our purposes, but we see this size as a proxy for both disk usage and compile time.


Figure 3.1: Histogram of the Repository Sizes. The threshold of 300MB is shown as dashed line. 23 outliers with a mean size of 1.46GB are not depicted.

4. **Maven Usage.** Compiling and executing repositories automatically requires all dependencies to be resolvable by a package manager. To simplify subsequent steps, we restrict our dataset

to repositories that use Maven[3] as package manager. To determine if a repository uses Maven, we check if there is a `pom.xml` file at the repository root, filtering out 46.9% of the repositories.

5. **Last Commit Date.** As a proxy for the quality of the test cases, we filter out stale repositories. Specifically, we consider a repository as stale if there were no commits on the main branch for the past two years. The threshold of two years was chosen intuitively because the distribution in Figure 3.2 did not reveal any distinct values that seemed particularly beneficial (note the long tail in the histogram and the almost linear decay in the cumulative sum). If mere maximization of the dataset size is desired in future work, increasing the threshold could a viable option.

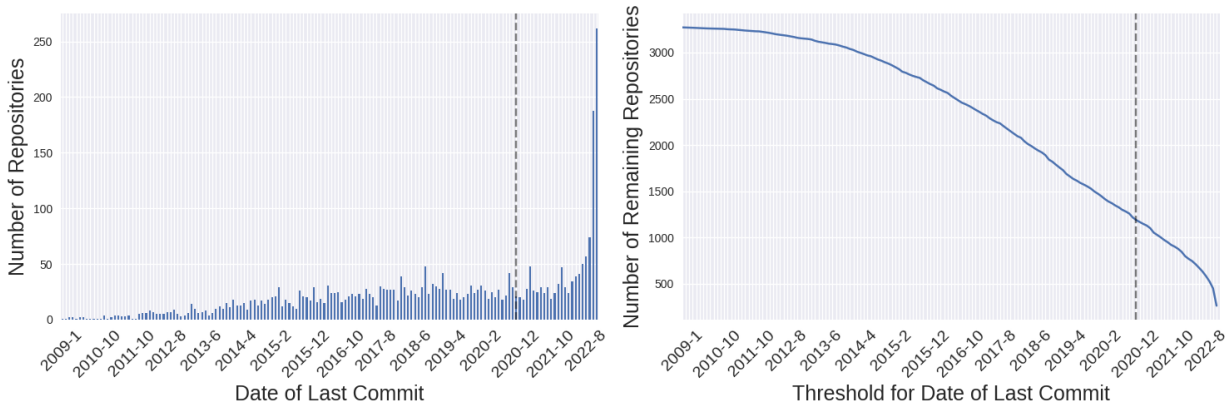



Figure 3.2: Histogram of last commit dates on the left. Number of remaining repositories for different threshold values on the right. Both include the chosen threshold of two years as a dashed vertical line.

6. **Number of Tokens.** After these first five filtering steps based on metadata, we clone the 1198 remaining repositories and use the mapping heuristic from Methods2Test [46] on them to get the pairs of MUT and test case. Since the dataset is meant for training and evaluation of language models, we restrict the combined length of MUT and test case to be at most 1850 tokens. This leaves 198 tokens for continuous prompts to the hard limit of 2048. For tokenization, we use the tokenizer of Polycoder. Note that in this step, we filter out individual test cases rather than complete repositories, unless all test cases exceed the token limit (0.4% of repositories). Figure 3.3 shows the distribution of the number of tokens, 96.5% of samples are below the threshold of 1850.

7. **Number of Test Cases.** We filter out any repository that has less than 80 mapped test cases in order to keep the number of repositories and the associated cost for compiling and executing them manageable. On the other hand, we want a diverse set of test cases that is not dominated by few repositories. Therefore, if more than 1000 test cases are mapped, we sample 1000 random ones. We find that there are many repositories with very few test cases, e.g., 8.6% contain only a single mappable test case and 74.4% contain less than 80 mappable test cases. Figure 3.4 displays the histogram of the number of test cases, as well as the cumulative sum.

[3] `https://maven.apache.org/`

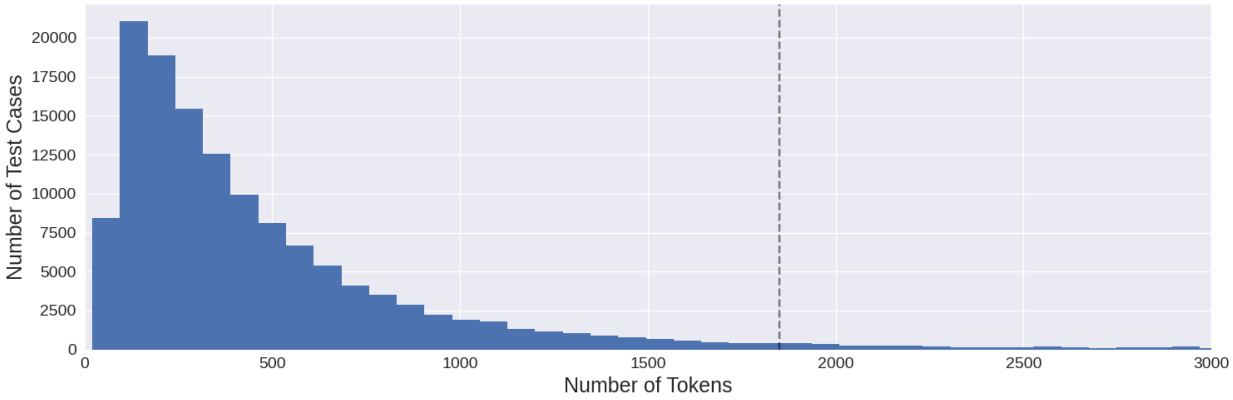



Figure 3.3: Histogram of the number of tokens of all mapped test cases. The threshold of 1850 is depicted as a dashed line. For readability purposes, 1,512 outliers with a mean of 4,816 tokens and a max of 36,862 tokens are not depicted in this figure.

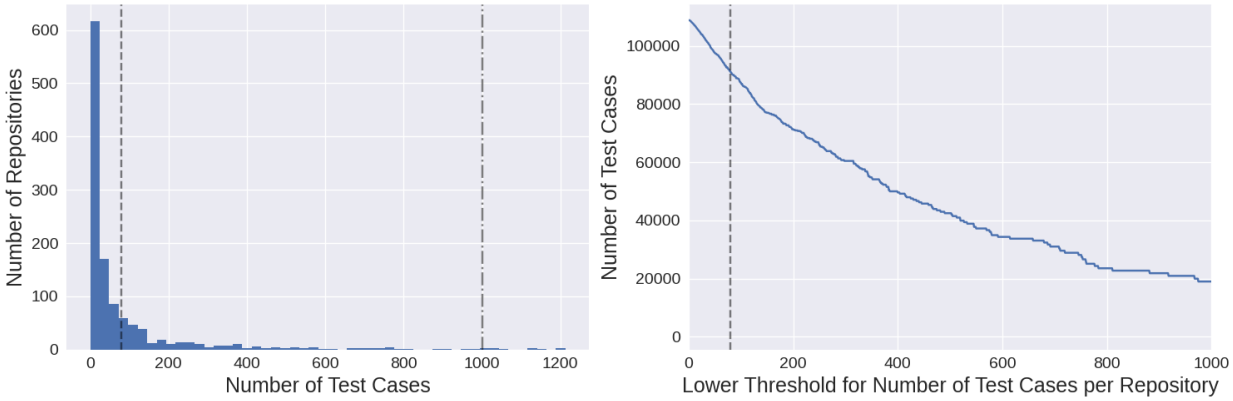



Figure 3.4: Left: Histogram of number of test cases per repository. The dashed line shows the lower threshold of 80, and the dot-dashed line represents the upper limit of 1000, above which we sample. Right: Plot of remaining total test cases for different lower thresholds, with the upper limit of 1000 already considered.

8. **Compilation.** We require all repositories to be compilable with maven, i.e. all dependencies must be specified in the `pom.xml` files. We first detect the Java version from the root `pom.xml`. We support Java 8, 11, 17 and 19, among which we select the closest most recent one. We run the compilation with `mvn clean compile` and keep the repository if the command succeeds. We find that roughly half of the repositories compile. We save the logs but don't further investigate reasons for compilation errors. Finding patterns in errors and adjusting the environment accordingly might yield more compilable repositories in future work.

9. **Execution.** Similarly, we require all repositories, or more specifically their test suites, to be executable in our environment. We select the Java version as described above, run `mvn clean test` and keep a repository if the command succeeds, eliminating 58.6% of the repositories. Similar to the compilation, we save the logs and leave further investigation to future work.

## 3.2 Dataset Split

After filtering the repository candidates, we split them into train, validation and test splits. These splits are created based on whole repositories, i.e., samples from the same repository will all be in the same split. This ensures that the benchmark measures generalization and avoids data leakage through duplicate or near duplicate code from the same repository [3]. We aim for a split where 70% of the test cases are in train, 10% in validation and 20% in test. Since the repositories contain different numbers of test cases, creating the split is not as trivial as assigning 70% of the repositories to train and so on. We therefore use the following procedure to create the split randomly, while also getting as close as possible to the target number of test cases. First, we calculate the target number of test cases per split. Next, we iterate over the repositories and randomly assign every repository to one of the splits that can include this repository without exceeding their target number of test cases. Afterward, there are a number of repositories left over. These leftover repositories are sequentially assigned to the split that is still farthest away from their target. Using this procedure, we obtain the distribution depicted in Table 3.3.

| | Number of Repositories | Number of Test Cases |
|---|---|---|
| Train | 41 (65.08%) | 8733 (69.93%) |
| Validation | 9 (14.29%) | 1285 (10.29%) |
| Test | 13 (20.63%) | 2471 (19.79%) |
| Total | 63 (100.0%) | 12489 (100.0%) |

Table 3.3: Number of Repositories and Test Cases per Dataset Split. Note that these numbers include the validation repository and 2 test repositories that were later removed due to incompatibility with the coverage tool stack.

## 3.3 Final Dataset

The final dataset consists of 63 repositories with ∼12k mapped test cases corresponding to ∼6k MUTs. Out of the 63 repositories, we remove 3 due to incompatibility with the coverage tool stack, as described in subsection 3.4.7. Table 3.4 displays more detailed statistics. We can observe that most MUTs are tested by only few test cases (1.87 on average). Most MUTs are rather short, for instance, 25% are shorter than 39 tokens, indicating low complexity. To confirm this, we further investigated the MUT complexity and found that 18.4% are getter methods, 2.7% are setter methods and the average cyclomatic complexity score [34] is 2.29. The dataset contains repositories using both Java 8 (43/63 or 68.25%) and Java 11 (20/63 or 31.75%). Moreover, the repositories use multiple test frameworks: JUnit4 (32/63 or 50.79%), JUnit5 (24/63 or 38.10%) and TestNG (7/63 or 11.11%).

| | Avg | Max | Min | 25% | 50% | 75% | Total |
|---|---|---|---|---|---|---|---|
| Number of Test Cases per Repository | 198.24 | 1000 | 80 | 102.50 | 146.00 | 244.00 | 12,489 |
| Number of MUTs per Repository | 106.03 | 672 | 7 | 57.00 | 78.00 | 129.50 | 6,680 |
| Number of Test Cases per MUT | 1.87 | 86 | 1 | 1.00 | 1.00 | 2.00 | 12,489 |
| Number of Tokens per MUT | 153.73 | 1681 | 5 | 39.00 | 80.00 | 182.25 | 1,026,888 |
| Number of Tokens per Test Case | 194.05 | 1668 | 13 | 79.00 | 136.00 | 235.00 | 2,423,516 |

Table 3.4: Aggregate Dataset Statistics. Token counts correspond to the PolyCoder tokenizer. 25%, 50% and 75% correspond to the respective percentiles.

## 3.4 Coverage Pipeline

To get coverage metrics for test cases generated by a model in a convenient way, we create a coverage pipeline that filters for executable test cases, inserts them into the repository and executes them. The main challenge is that model-generated test cases can be arbitrary, i.e., we do not have any guarantee of parsability, compilability or executability. Therefore, the pipeline sequentially filters out faulty test cases in multiple steps, such that only executable test cases remain in the end. Figure 3.5 gives an overview of the pipeline. In the following, we describe these steps and their respective challenges in more detail.

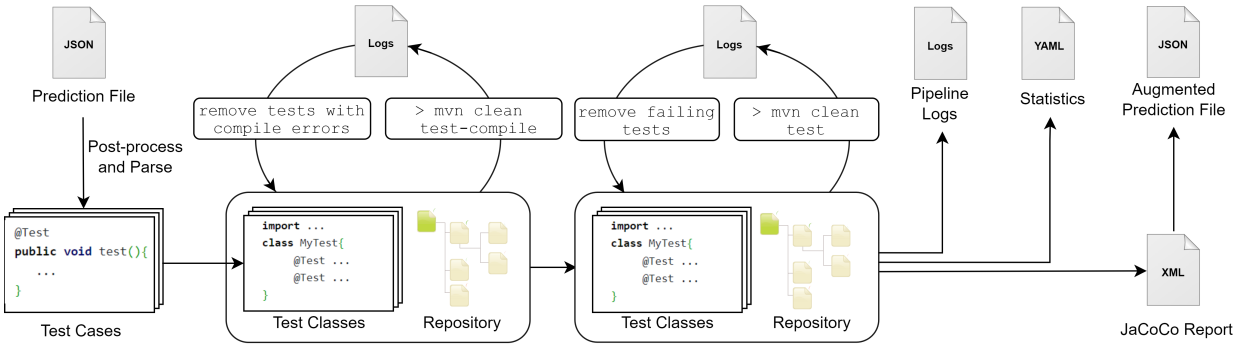



Figure 3.5: Overview of the Coverage Pipeline

### 3.4.1 Post-processing and Parsing

The first step is to apply a lightweight post-processing and filter for code that is parsable. The main case that the post-processing covers is incompleteness: the LM might not have completed the test case at the token limit or be stuck in an infinite loop. We first truncate everything after the closing outer parenthesis and attempt to parse the result with javalang[4] using the `parse_member_declaration` method. We assert that the parsed tree represents a method declaration, rather than a class declaration for example. If parsing fails, we truncate back to the last complete statement, indicated by the last semicolon. Afterward, we deduplicate statements at the end of the method to fix infinite loop cases and close all open parentheses. Finally, we attempt to parse again and discard the failing samples. Since the goal of this work is to evaluate the proposed modelling technique, we keep the

[4] https://github.com/c2nes/javalang

post-processing rather simple, but more sophisticated repair techniques could be incorporated in future work.

After repairing and parsing test cases, we modify its annotations. First, we remove potential `@Ignore` and `@Disabled` annotations to ensure that the test case will be executed later on. Second, we configure an execution timeout of 10 seconds per test case. To that end, we detect the test framework (JUnit4, JUnit5 and TestNG) based on the import statements in the test class and modify the annotations to include the respective timeout settings. In particular, we add `@Test(timeout=10000)` for JUnit4, `@Test(timeOut=10000)` for TestNG and `@Timeout(10)@Test` for JUnit 5.

### 3.4.2 Repository Configuration

Before attempting to compile the parsable test cases, we prepare the repositories to create a standardized environment. For that purpose, we first copy the repository to a temporary directory and perform all modifications there. That way, the original repository state remains intact for future changes of the coverage pipeline. This might be omitted in the future to avoid the slight overhead of creating the copy and modifying the configuration on every execution of the pipeline.

In Maven, arbitrary plugin executions can be hooked into the test phase triggered by calling `mvn test`. The purposes of such executions are manifold. Some plugins are required to build the project, while others are mainly supporting developers with analyses or deployments. In the context of our benchmark, we want to limit the execution to those plugins necessary for compilation to accelerate the process and avoid failures due to errors unrelated to our use case. Through manual investigation, we find `maven-compiler-plugin`, `jaxb2-maven-plugin`, `build-helper-maven-plugin`, `antlr4-maven-plugin`, `maven-jar-plugin`, `protoc-jar-maven-plugin`, `maven-bundle-plugin`, `maven-shade-plugin` and `maven-install-plugin` to be the essential plugins in our selection of repositories. They either configure the compilation and packaging process or generate code. We keep these plugins and remove all others from all `pom.xml` configuration files.

To create a uniform environment that allows automatic coverage evaluation for different repositories, and to enable coverage analyses, we programmatically modify the project configuration. The first challenge is to find the `pom.xml` file that is a common ancestor to all sub-configurations, such that the modifications take effect for all submodules. While very common, the root configuration does not have to be the `pom.xml` file at the root directory of the project. Therefore, we traverse the tree created by the parent relations between configurations until we arrive at the root. This does also not necessarily yield a single configuration file, i.e., the relations can sometimes constitute multiple unconnected trees. In our selection of repositories, this only happens when artifact or resource directories also contain `pom.xml` files, which do not need the common configuration options. Consequently, we select the `pom.xml` file with the most children as a heuristic, which works for all our repositories. After selecting the root configuration that influences all sub-configurations, we make the following modifications.

1. When filtering for compilable test cases among a large number of generated test cases, it is essential to receive as many compilation errors as possible, such that the faulty test cases can be removed. To achieve that, we configure the `maven-compiler-plugin` not to stop the

compilation process when errors occur and increase the maximum number of displayed errors (see appendix A.1.1). Note that this plugin is also among the essential plugins we identified. Therefore, we merge the original settings with these new ones.

2. Next, we configure the `maven-surefire-plugin`, which controls the test execution. Similar to the compilation step, we instruct the plugin to keep running even when errors occur, to get as much feedback as possible in one run (see appendix A.1.2).

3. Lastly, we add the configuration for the `jacoco-maven-plugin` (see appendix A.1.3). It is used to measure the code coverage. We chose JaCoCo[5] over alternatives because it is the most mature tool that supports most Java versions.

### 3.4.3 Scaffolding

We view the test case generation task on the method level, yet methods alone cannot be compiled, but need a test class. A complete tool would generate a scaffolding, including all imports and a test class. While not impossible, generating this automatically is not trivial for many reasons, e.g., it is not always clear from which package a name should be imported. Since this work focuses on the method level, we instead use the test classes of the ground truth test cases as scaffolding. In all cases except one, there is only a single test class for every MUT in the dataset.

### 3.4.4 Compilation

The goal of this step in the pipeline is to identify test cases that are not compilable. To that end, the compiler was configured to continue compiling on errors and to output all errors. Ideally, the compiler would identify all compilation errors in one run. In practice, this is not realistic due to masking effects between errors, i.e., some errors only occur when others are resolved. Therefore, we iteratively compile the project and remove faulty test cases. Specifically, we compile using `JAVA_HOME=/path/to/java/version mvn clean test-compile`, where the appropriate java version is detected from the root configuration. Next, we parse the logs produced by the compiler using regular expressions. Unfortunately, the errors are not always formatted consistently. The three regular expressions in appendix A.1.4 cover all variants that we experienced during the experiments and extract the file name, line number of the error in that file and the error message. We then parse the test classes into their Abstract Syntax Trees (ASTs), find all test cases that span across at least one of the error lines, and remove these test cases from their test class. We iterate until no further errors are detected by the regular expressions, such that all remaining test cases are compilable. We save both compilation logs and error messages per test case for post-hoc analyses.

### 3.4.5 Execution

Similar to the compilation step, one goal of executing the test cases is finding those that run successfully. We again construct regular expressions to parse the logs, extract the files and names of the failing test cases, and remove them from their test classes. Unlike during compilation,

[5] https://github.com/jacoco/jacoco

removing failing test cases would not be strictly necessary to remove failing test cases. However, parsing the logs yields interesting information about how many and which test cases were ultimately runnable. At the same time, by removing the failing test cases, we make sure that only passing tests contribute to the final coverage metrics. We save all error information, e.g., enabling subsequent analyses of assertion errors.

### 3.4.6 Report Parsing

The execution of `mvn test` produces the JaCoCo reports as artifacts. We first parse the XML reports and get the coverage information for all methods. We then retrieve the coverage for all MUTs by matching their signature to the JVM signatures in the JaCoCo report. To that end, we leverage the `org.jacoco.report.JavaNames` class to convert the JVM signatures to a more easily readable and parsable format. Ultimately, we extract both line and branch coverage per MUT and aggregate.

### 3.4.7 Pipeline Validation and Manual Corrections

To validate the correctness of the pipeline, we run it in two settings, where we know the expected outcome. First, we implement a dry run where the pipeline is executed without inserting any test cases into the repositories. Therefore, in that setting, we eliminate all steps that are concerned with processing the test cases as sources of error. We ensure that all dependencies are available and make sure that the project is compilable without non-essential plugins, and therefore validate that our list of essential plugins is sufficient. Furthermore, we check that no test cases are executed, as all test cases should have been removed from the repository during the scraping process.

Second, we further leverage the ground truth test cases for validation purposes. In other words, we re-insert the mapped test cases into their test classes and attempt to measure their coverage. To that end, we first transform the mapped test cases from the dataset format into the prediction format, such that the whole pipeline is executed as if the test cases were model-generated. Ideally, we expect all test cases to be parsable, compilable, and runnable. We find that 99% of test cases are executable, validating the pre-processing, compilation and execution procedure. We further find that 98% of those also test the correct MUT, validating the mapping heuristic. We analyze the few test cases that are not parsable, compilable or executable and find three patterns of edge cases. First, some test cases depend on the side effects of other test cases, like setting attributes in shared objects. These test cases might fail in our setup because not all test cases of original test classes are mapped during scraping and because the execution order can differ. Second, some test cases exceed the 10 seconds timeout. Lastly, some test cases fail with various error types after their `@Disabled` or `@Ignore` annotations were removed. This is likely because these test cases are outdated with respect to their MUT. We conclude that the pipeline works correctly and that the few edge cases are not due to a bug in the pipeline.

A high degree of automation was one of the main goals throughout the benchmark creation, to manage and support the large scale of this study. At the same time, it makes the dataset extensible for future work. In that sense, the scraping process is fully automated, and a larger dataset can be

created by simply adding more candidate repositories. However, when adding a new dataset to the pipeline, one should manually validate that the repository works in the pipeline environment. For instance, one should ensure that there are no dependency conflicts between the repository and the pipeline tooling. We believe that the two validation approaches discussed above are also excellent test cases to do that. For the 22 repositories in the validation and test set, we perform this manual validation. In the following, we list all manual changes we made to these repositories, that were identified in the process:

1. `intuit/CloudRaider` uses `powermock`[6] in their test classes to create mock objects. Unfortunately, `powermock` uses on-the-fly code instrumentation and is therefore incompatible[7] with JaCoCo's on-the-fly instrumentation. For that reason, we exclude this repository from any further experiments. We further exclude `Flipkart/foxtrot` because it depends on running database containers and `SonarSource-orchestrator` because its build process is incompatible with JaCoCo.

2. The configuration of `bazaarvoice/ostrich` has a parent configuration outside of the repository itself, that overrides the `argLine` property of surefire. This override breaks JaCoCo, so we remove the dependency to that parent configuration.

3. In `eclipse/winery`, we find that the `frontends` submodule has additional dependencies to NodeJS and takes a very long time to compile. At the same time, it does not contain any mapped test case and no other submodule depends on it. We therefore remove this submodule from the build by removing the corresponding `<module>` entry in the root configuration.

4. `JUnit5` did not support timeouts until version `5.5`. Therefore, we upgrade any older version to `5.5.1`. This is the case for `Domo42/saga-lib` and `flipkart-incubator/Lyrics`. We further upgrade `JUnit4` from `4.11` to `4.12` in `Domo42/saga-lib` in order to make it compatible with `junit-vintage-engine` (which was used in the repository all along) and therefore to ensure that all test cases are executed.

5. During scraping, we removed all test cases from the repositories, such that only the inserted test cases would contribute to the coverage. The script found and removed test cases annotated with `@Test` and `@ParameterizedTest`, but not those that use the JUnit3 style of declaring a class as a test class: by extending it from `junit.framework.TestCase`. This occurred only a single time and was corrected manually, but could be automated in the script in a future iteration.

## 3.5 Evaluation Metrics

For evaluation, Tests4J first computes BLEU and crystalBLEU scores as intrinsic metrics. These metrics compare generated test cases to correct, human-written test cases for the same MUT. In particular, they compute scores based on n-gram overlaps. CrystalBLEU ignores trivial n-grams

[6] https://github.com/powermock/powermock
[7] https://github.com/powermock/powermock/wiki/Code-coverage-with-JaCoCo

in that process and thus more accurately measures code similarity. Both can give an indication of the quality of a test case, but don't capture its behavior during execution. Therefore, the pipeline computes several metrics, including four coverage metrics: *MiLC*, *MaLC*, *MiBC* and *MaBC*. They correspond to the micro (*Mi*) and macro (*Ma*) averages of the line coverage (*LC*) and branch coverage (*BC*). The micro average is the proportion of the total lines or branches of the MUTs that were covered by the tests, whereas the macro average is computed per MUT and then averaged. Therefore, *MiLC* and *MiBC* account for the fact that MUTs vary in complexity and weighs methods with many lines or branches higher than those with few. Consequently, we use them as main coverage metrics. We additionally report *MaLC* and *MaBC* because, in comparison with their micro average counterpart, they indicate whether the test suite covers mostly small or large MUTs. Besides coverage, we further report five count metrics that quantify how many test cases remain at certain stages of the pipeline:

1. *Unique*: Number of test cases after de-duplication.
2. *Parsable*: Number of test cases that comply with the Java grammar, absolute and relative to *Unique*.
3. *Compilable*: Number of test cases that were compiled in their scaffolding without error, absolute and relative to *Unique*.
4. *Executable*: Number of test cases that were executed without errors, absolute and relative to *Unique*.
5. *Correct*: Number of test cases that were executed without errors and called the correct MUT, absolute and relative to *Unique*. This metric is adopted from Tufano et al. [47] and is directly comparable, albeit computed on different repositories.

An example of all metrics can be found in Table 3.5, which displays the results of the ground truth test cases.

## 3.6 Ground Truth Results

Table 3.5 displays the results of inserting the mapped test cases (called "ground truth test cases") back into their respective test classes and evaluating their coverage with the pipeline described above. Besides being used for the pipeline validation as discussed in subsection 3.4.7, the results yield some further insights regarding the dataset. For instance, the ground truth test cases achieve very high coverage, even though there are only 1.87 test cases per MUT on average with an average length of 194 tokens. We conclude that the MUTs are usually testable with relatively few and short test cases, i.e., there usually is a good and simple solution. We also observe that for both line and branch coverage, the micro average is higher than the macro average. Since the micro average is equivalent to a macro average weighted by total lines, this indicates that the ground truth tests were more successful in covering complex MUTs than simple ones. This is opposed to test cases generated by deep learning-based approaches, where typically the macro average is higher. Further investigations are necessary to find the reasons for this opposing trend. The BLEU scores are obviously very high because we compare the ground truth test cases to themselves, but they are not 1 because of the preprocessing.

Ground Truth BLEU: 0.9455 crystalBLEU: 0.9369

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 83.54% | 86.65% | 76.28% | 85.59% | 109 | 109 (100.00%) | 107 (98.17%) | 106 (97.25%) | 102 (93.58%) |
| bazaarvoice-ostrich | 11.50% | 11.27% | 10.32% | 10.92% | 158 | 157 (99.37%) | 157 (99.37%) | 157 (99.37%) | 153 (96.84%) |
| adorsys-XS2A-Sandbox | 93.98% | 96.03% | 85.51% | 92.20% | 180 | 180 (100.00%) | 180 (100.00%) | 178 (98.89%) | 178 (98.89%) |
| flipkart-incubator-Lyrics | 97.30% | 99.02% | 81.75% | 93.89% | 102 | 102 (100.00%) | 102 (100.00%) | 102 (100.00%) | 102 (100.00%) |
| seedstack-seed | 84.43% | 89.00% | 74.61% | 84.30% | 97 | 97 (100.00%) | 97 (100.00%) | 97 (100.00%) | 86 (88.66%) |
| intuit-QuickBooks-V3-Java-SDK | 70.19% | 75.29% | 59.55% | 75.53% | 238 | 238 (100.00%) | 238 (100.00%) | 237 (99.58%) | 237 (99.58%) |
| gooddata-gooddata-java | 77.44% | 87.48% | 83.58% | 88.34% | 377 | 375 (99.47%) | 375 (99.47%) | 375 (99.47%) | 374 (99.20%) |
| Flipkart-foxtrot | - | - | - | - | - | - | - | - | - |
| shroffk-phoebus | 74.98% | 87.02% | 67.39% | 81.06% | 285 | 285 (100.00%) | 279 (97.89%) | 277 (97.19%) | 274 (96.14%) |
| messaginghub-pooled-jms | 87.86% | 93.42% | 93.87% | 97.72% | 239 | 239 (100.00%) | 239 (100.00%) | 239 (100.00%) | 239 (100.00%) |
| nhl-dflib | 84.64% | 77.94% | 73.71% | 72.18% | 134 | 134 (100.00%) | 134 (100.00%) | 134 (100.00%) | 129 (96.27%) |
| monarch-initiative-phenol | 86.68% | 86.32% | 76.92% | 82.05% | 120 | 120 (100.00%) | 120 (100.00%) | 120 (100.00%) | 116 (96.67%) |
| SonarSource-orchestrator | - | - | - | - | - | - | - | - | - |
| Total | 78.32% | 68.42% | 71.96% | 66.44% | 2039 | 2036 (99.85%) | 2028 (99.46%) | 2022 (99.17%) | 1990 (97.60%) |

Table 3.5: Evaluation results of re-inserting the ground truth test cases back into their test classes.

# 4 Context-Aware Prompt Tuning

In this chapter, we propose a novel method we call Context-Aware Prompt Tuning. It is fundamentally motivated by the need for more contextual information, as well as the limited context length and quadratic complexity of transformer language models. We further discuss this motivation in section 4.1 and present Context-Aware Prompt Tuning in section 4.2.

## 4.1 Motivation

One of the fundamental premises of this work is that contextual information is important for test case generation. In the most basic setup of neural test case generation, the model translates from a MUT to a test case. However, we argue that this task setup is ill-posed because the input is missing important information, without which not even a human expert developer could write correct test cases. For instance, writing tests requires knowledge about available classes, methods, fields and libraries from the current project. Whereas it is likely that models are able to use the APIs of popular libraries due to their usage in the pre-training corpus, it is unlikely to impossible that the model is able to infer APIs of the local project. Fine-tuning the model to a specific project [8] and saving this local knowledge in the model parameters is possible but not feasible for many evolving projects. Instead, contextual information can be passed to the model in the prompt. The model is then not trained on a specific project, but simply trained to leverage the contextual information given in the prompt. Tufano et al. [47] investigated the effect of additional context for the test case generation task and were able to reduce the loss by almost 10%. Similarly, the effects of context from the current project have been shown to significantly improve performance on other code-related tasks as well [44, 53]. Specific to our benchmark is that contextual information is not only required from the focal class and potentially other classes in the main code, but also from the test class. The evaluation pipeline directly inserts generated test cases into a fixed scaffolding. This scaffolding already contains imports and the test class declaration. This test class often contains declared fields, setup code and helper methods. Generating correct test cases that fit into this test class obviously requires this test class knowledge, otherwise available names and signatures would need to be inferred or hallucinated by the model.

To summarize, in this task setup, a model has to process at least the MUT, the focal class as well as the test class, and infer a correct test case from that information. With the token limit of 2048 in PolyCoder, this is not possible. The complete information would require more than 7000 tokens on average. To mitigate the issue, prior work [47] has suggested omitting method bodies and only including their names and signatures, with the intuition that this already provides the

most important information on how to use it. Whereas this approach reduces the total number of tokens to about 1900 on average, more than 16% of the samples would still exceed the hard limit of 2048 tokens. Furthermore, maxing out the token limit might not be desirable because of the quadratic memory and compute complexity with respect to the sequence length on transformers. We illustrate this in a toy example and measure the memory consumption of training the smallest PolyCoder model with just 160M parameters in 16-bit precision with a batch size of 4. Figure 4.1 presents the results of that experiment. Even in this minimalistic setting, a 32GB GPU is maxed out when approaching the maximum sequence length.

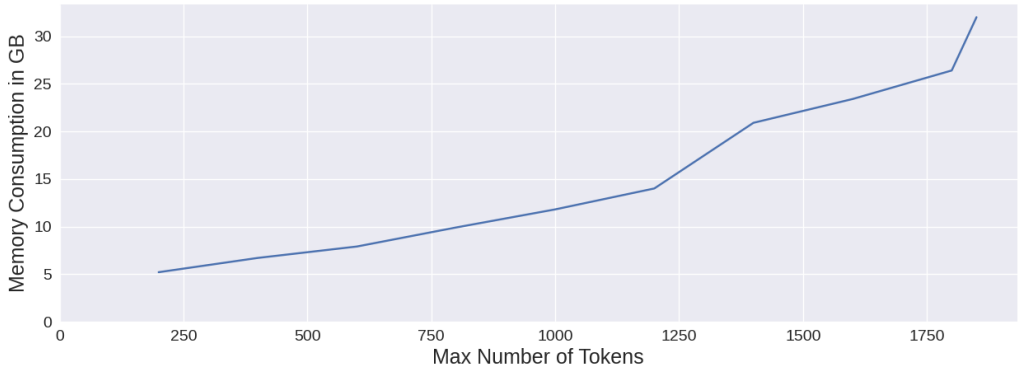



Figure 4.1: GPU memory consumption during training of PolyCoder-160M in FP16 with a batch size of 4 and varying sequence lengths.

## 4.2 Method

Motivated by the issues discussed above, we propose Context-Aware Prompt Tuning (CAPT), a novel method to allow the model to ingest more context without quadratic scaling of memory requirements. The idea is fundamentally based on soft prompt tuning, where the model learns continuous embeddings representing soft tokens that steer a frozen pre-trained LM towards performing a specific task. In CAPT, we aim to learn soft tokens that contain particular context information and thus steer the model towards generating test cases that better fit that context. To that end, we use small neural networks that first summarize and compress the context information into one or multiple soft tokens. We call these networks Prompt Generators (PGs). They take a piece of context as input and output an embedding that is then injected into the LM. Unlike other methods [44], the output of the Prompt Generator (PG) is continuous, which allows joint end-to-end training. I.e., we backpropagate the gradients from the language modelling loss through the language model to the PGs. Figure 4.2 presents an overview of the approach.

### 4.2.1 Context Types

The proposed method of encoding context with a PG into an embedding allows for flexible context modalities. Much like in [24], the context does not need to be of textual nature, but could be images or videos. For test case generation, code is arguably one of the most important modalities.

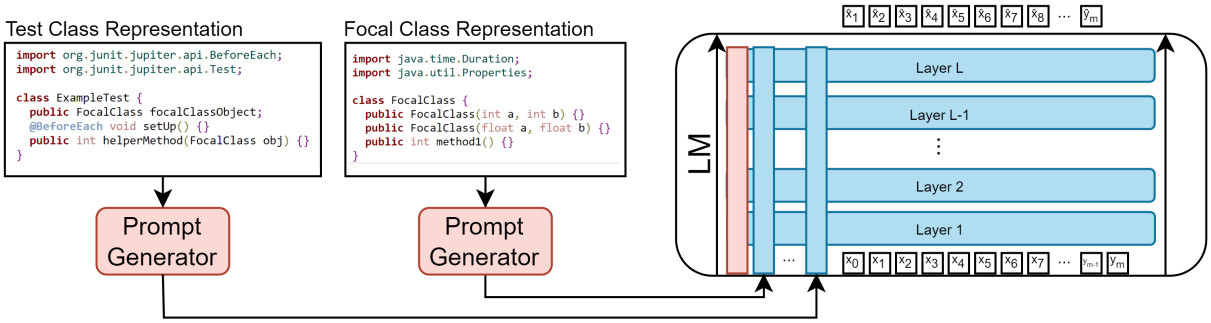



Figure 4.2: Context-Aware Prompt Tuning with test class and focal class context encoded as soft tokens with an all-layer injection strategy. The left-most prefix is a vanilla (instance-invariant) task prefix.

However, others could be beneficial as well. For instance, class hierarchies and call graphs could be encoded with graph neural networks and provide valuable information about the overall structure of the project and natural language description from documentation and Javadoc could provide low-level semantic information about classes and methods. In this work, we focus on code as modality and investigate the possible compression rather than multi-modality. As explained in section 4.1, the focal class and test class are the two most important sources of information. Following previous work [47], we omit method bodies and try to only include the most relevant information:

1. **Imports** are relevant for the test class context because they introduce names that are available in the test case to generate. They are not relevant for the focal class because the model does not need to generate a method in its namespace.

2. The **class name** is relevant for multiple reasons. In the focal class, it provides semantic clues about the class functionality and gives the model partial information on how to instantiate an object of that class. In the test class, it mainly contributes to a coherent context, as providing syntactically incorrect code might confuse the model.

3. The **method signatures** are equally important for focal classes and test classes. First, they give semantic clues about the class, inform the model which methods are available and how to use them. To maintain syntactic correctness, we do not simply remove the method bodies but replace them with an empty block `'{}'`. In focal classes, we only include public methods. In test classes, we do not remove the body of the setup method marked by the `'@Before'` annotation. We argue that knowing the state of the objects instantiated in these setup methods is very important, especially for generating correct assertions, and that the setup code can provide this information to a high degree.

4. Lastly, the **fields** also provide both semantic clues and information on how to use the focal class. In test class, they inform the model on the declared names and their types. As with methods, we only include public fields for the focal class context.

To optimally leverage knowledge from pretraining, we format this context in a way that should look as natural as possible to the model. We provide further details on the formatting and truncation of context in the experimental setup in subsection 5.1.1.

### 4.2.2 Prompt Generator Architectures

The prompt generators (PGs) are the fundamental building blocks of CAPT. They compress contextual information in order to steer a LM. In this section, we first sketch the design space for PGs and afterward derive 6 concrete architectures from that space. Every prompt generator takes a context as input and outputs one or multiple soft tokens. In the following, we list the design decisions that we considered when designing prompt generators.

- A very fundamental decision is how to represent the context. Various representations like graph-based representations are possible, but for the sake of simplicity, we focus on **token-based representations** in this work.

- Next, the tokens need to be embedded. Here, we focus on lightweight approaches based on pre-trained models. Using the main LM itself to create **context embeddings** would likely yield more powerful representations, but it would also cause significant compute and memory overhead to perform multiple forward passes and thus defeat the purpose of CAPT. Instead, we consider two approaches: using the input embeddings of PolyCoder and using CodeBERT.

- After obtaining an embedding for every token, these embeddings need to be projected into the target space. This **target space** is either the input space of the LM (1024 dimensional) or the space of prefix tokens across all layers ($1024 * 24$ layers $* 2 = 49,152$ dimensional). This depends on the way the soft tokens are injected into the LM, either only at the input layer as in Prompt Tuning [29] or at every layer as in Prefix Tuning [31] and P-Tuning v2 [33]. We find that both approaches work similarly well. Since at the time of implementation, passing input embeddings to a LM was not supported during generation[1], we focus on injecting soft tokens at every layer. The concrete implementation of how to project token embeddings into the target space also requires a few further architectural decisions. In particular, we perform two main steps: aggregation and projection.

- During **aggregation**, multiple token embeddings are combined into a fixed-size representation. That is, the aggregation step performs the actual compression of the technique. Overall, we consider four aggregation approaches: CLS token, sum, LSTM and MLP. Whereas CLS token, sum and LSTM can aggregate variable-length sequence into a fixed-size representations, an MLP requires a fixed-length input.

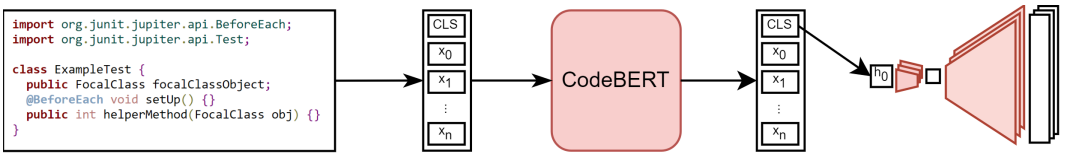
- During **projection**, the fixed-size representation from the aggregation step is projected into the target space. We consider a simple linear layer, as well as a 2-layer MLP with a bottleneck. In the former, the first layer projects the aggregated embedding down to an even smaller embedding, followed by a non-linear activation function, before the second layer projects it up into the target space. Bottleneck are also used in regular prefix tuning, and we can confirm in preliminary experiments that bottlenecks improve performance slightly. We therefore use bottlenecks in all setups.

[1] Support was recently added in v4.27: `https://github.com/huggingface/transformers/issues/6535`

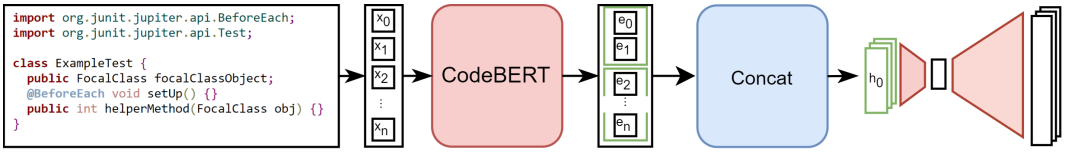
- Lastly, an important consideration is how to scale the number of soft tokens that the prompt generator outputs. We see the concept of compression at the center of CAPT. Thus, we believe being able to flexibly change the rate of compression is crucial. The amount of information that can be stored in an embedding is limited, forming an **information bottleneck**. We implement two main ways of scaling. First, we use the same aggregated context representation for all tokens, but use separate projections. The major drawback is that all soft tokens are based on the same information and thus scaling the number of soft tokens will likely not increase the performance much. Motivated by this reasoning, we also propose to use chunked aggregation, where the token embeddings are chunked before aggregation. Then each aggregated chunk embedding is projected into the target space. Thereby, every soft token is based on the information from different parts of the context.

Based on these general concepts and design consideration, we now present 6 prompt generator architectures. In the architecture visualizations, blue indicates frozen modules, red indicates trainable modules, green boxes represent chunks and the right-most boxes with black borders represent the soft tokens that will be injected into the LM. We use trapezoids to illustrate linear projections that alter the dimensionality. All architectures use `tanh` as non-linear activation function in the bottleneck. We indicate models that chunk the context with the suffix $\_C$.

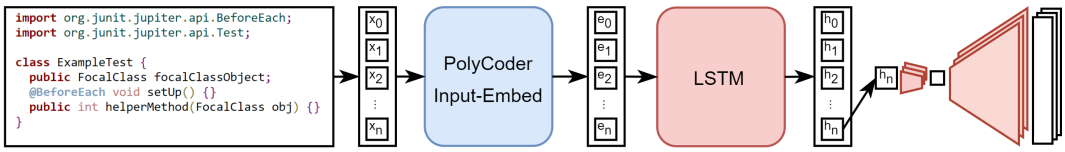
1. In **BERT**, we pass the context through CodeBERT [18] and use the embedding of the CLS token as a holistic representation of the context. This representation is then fed through a projection MLP with bottleneck to obtain a soft token. To create $k$ soft tokens, the CLS representation is projected with $k$ different MLPs. With a bottleneck dimension of 512, the number of trainable parameters amounts to 110M in CodeBERT and $k * (768 * 512 + 512 * 49,152)$ in the projection modules, totaling $\sim$ 135M parameters for 1 soft token and $\sim$ 365M parameters for 10 soft tokens. Therefore, the number of parameters scales linearly with $k$, making it only a viable option if $k$ is small. Furthermore, all information has to be contained in the CLS token, such that increasing $k$ would likely not result in significantly better results.



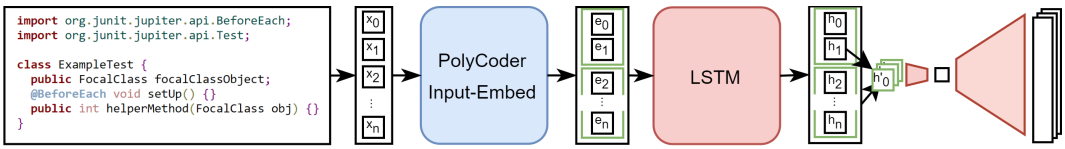

2. Contrarily, in **BERT_C**, we divide the sequence of context tokens into fixed-size chunks and concatenate the embeddings of all tokens in each chunk. The chunk size is determined based on the number of soft tokens $k$ as $cs = floor(512/k)$. We project the chunk embeddings into the target space with a bottleneck of 1024. Since each of the $k$ soft token is based on different parts of the context, we share the projection module among them. Consequently, the number of trainable parameters is independent of $k$ and amounts to 110M for CodeBERT and $cs * 1024 + 1024 * 49,152 \approx cs * 1024 + 50$M. We expect this architecture to have better scaling behavior than BERT because increasing $k$ will make the chunks smaller, thus decreasing the compression rate and alleviating the information bottleneck.



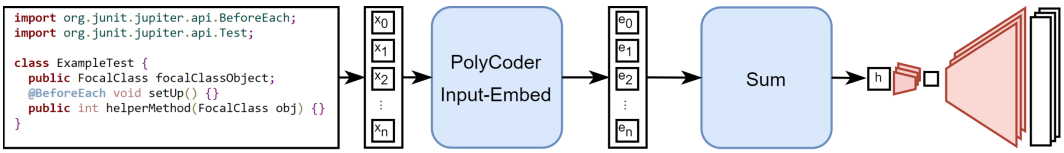

3. In **LSTM** and all further architectures, we use the input embedding layer of PolyCoder to embed context tokens. This approach is much more lightweight than CodeBERT, but has the potential advantage that it uses the same tokenizer and embedding space as the LM. In this variant without chunking, we process the complete context with a single-layer LSTM and use the last token embedding as a holistic representation. The projection layer is equivalent to the one described for BERT. In that sense, the setup also shares its drawbacks of the limited scaling possibilities and the small information bottleneck. The number of trainable parameters is $8 * 1024 * 1024 \approx 8\text{M}$ for the LSTM and $k * (1024 * 512 + 512 * 49,152) \approx k * 25\text{M}$ for the projection module.



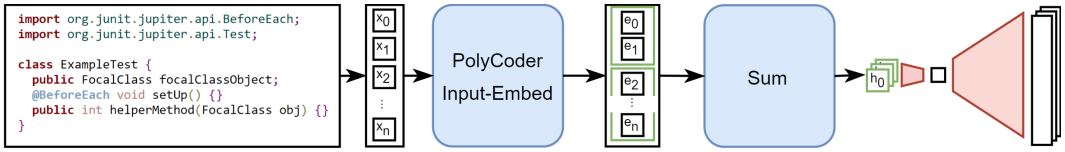

4. In **LSTM_C**, we use the same setup as in LSTM but chunk the context tokens into $k$ chunks, process each chunk with an LSTM separately and use the embedding of the last token of each chunk as chunk embedding. As with BERT_C, the projection module is shared among the $k$ soft tokens and increasing $k$ will alleviate the information bottleneck. The number of trainable parameters is independent of $k$ and totals about 33M.




5. In **Sum**, we also use PolyCoder's input embeddings but use the sum as non-parametric approach to aggregation. In this non-chunked variant, the sum of all token embeddings is used as a holistic representation and projected into the target space with the same projection module as in LSTM and BERT. The number of trainable parameters is $k * (1024 * 512 + 512 * 49,152) \approx k * 25\text{M}$.




6. Lastly, propose the chunked variant of Sum, **Sum_C**, where the embeddings of all tokens in a chunk are summed to form the chunk embedding. The projection module is the same is in LSTM_C and BERT_C. The number of trainable parameters is $\sim 25\text{M}$.


import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
class ExampleTest {
public FocalClass focalClassObject;
@BeforeEach void setUp() {}
public int helperMethod(FocalClass obj) {}
}
x0
x1
x2
xn
PolyCoder
Input-Embed
e0
e1
e2
en
Sum
h0

# 5 Experiments

In this chapter, we describe the experiments that we performed to investigate five research questions (RQs). We first describe the experimental setup in section 5.1. Afterward, we define the RQs and introduce the experiments we designed to answer each RQ in section 5.2. Finally, we present the results in section 5.3.

## 5.1 Experimental Setup

In the following subsections, we provide details about pre-processing, training, generation, as well as evaluation metrics.

### 5.1.1 Pre-processing

During pre-processing, we tokenize the different parts of the input and assemble them. In all experiments, we use a maximum sequence length of 1850, which leaves room for 198 potential soft tokens. We always perform padding to the longest sequence in each batch and truncate to the maximum sequence length. As elaborated on in subsection 4.2.1, we identified four essential components of focal and test classes: imports, class name, method signatures and fields. During pre-processing, we want to make sure that the condensed context representations look as much as possible like real code. To that end, we replace method bodies with empty blocks `'{}'` instead of just removing them. We place MUTs inside their focal class and test cases in their test class, and account for indentation. We furthermore argue that truncating the context naively will likely result in incomplete classes and statements, negatively conditioning the LM. We therefore implement a truncation procedure, where if the context is too long, we successively remove parts of the context according to a prioritization without sacrificing syntactic correctness. In particular, we go through the steps listed below, check if the context is short enough and only continue to the next step if it isn't. We define this prioritization for the scenario of fitting both test class and focal class context into a fixed token budget, which is the case when using both contexts as hard prompt. In other scenarios, we skip the steps that work on the respective other context. When using both contexts, we generally prioritize test class context over focal class context based on the insights of RQ2.

1. We first remove the fields of the focal class and thus assign it the lowest priority, based on the intuition that fields in the focal class are mostly relevant for the inner workings of the class and that usage is defined via interfaces and methods.

2. Next, we remove the method signatures of the focal class. However, we keep constructor signatures, given their special importance in test cases.

3. When the focal class representation is reduced to its minimum (class name and constructor signatures) and the context still exceeds its token limit, we remove the method signatures of the test class. We argue that compared to imports and fields, using helper methods is useful but not necessary for test cases.

4. Afterward, we remove the imports of the test class with the intuition that imports and fields convey similar information in the test class (which names are available), and that imported names are easier to infer than field names.

5. Lastly, we remove the fields of the test class, leaving minimal representations for both focal and test class context. If they still exceed the token limit (occurs $< 1\%$ of the cases), we truncate naively.

An example of the formatting and visualization of this prioritization can be found in Figure 5.1.

### 5.1.2 Training Procedure

For all neural models, we use the PolyCoder model [52] with 400M parameters. The PolyCoder family of models is publicly available, has good performance and is, unlike CodeGen [35], transparent with respect to the pre-training corpus. The variant with 400M parameters allows more direct comparison with AthenaTest [47] and enables faster experimentation compared to the 2.7B variant. In prefix tuning, we use 10 soft tokens, as that performs best in preliminary experiments. For optimization, we use regular cross-entropy loss and the `AdamW` optimizer from PyTorch. We set the learning rate to $5e-6$ for prefix tuning and to $1e-6$ for full fine-tuning. We train for 3 epochs with a batch size of 8 and evaluate on the validation set 4 times per epoch and save a checkpoint. The checkpoint corresponding to the best validation loss is used for testing.

### 5.1.3 Generation Procedure

We sample from the trained models with `top_k=50` and `top_p=0.9`. Furthermore, we increase the temperature from 0.05 to 0.4 in increments of 0.05. We generate 4 samples per temperature, i.e., a total of 32 samples per MUT. In preliminary experiments, we found this setting to provide a good tradeoff between diversity and quality. These experiments also showed that increasing the number of samples can significantly improve the coverage, e.g., increasing it from 32 to 96 increased it by almost 50%. We use the moderate number of 32 samples to save computational resources and to stay somewhat comparable to AthenaTest, which uses 30 samples. The post-processing is performed as described in subsection 3.4.1.

### 5.1.4 Evaluation metrics

As described in section 3.5, we calculate the four coverage metrics MiLC, MaLC, MiBC and MaBC from the coverage reports generated by Tests4J. Among these, we use MiLC, the fraction of total

```java
// Focal Class:
public class SwingListbox extends AbstractSwingContainer
                          implements XulListbox, ListSelectionListener  {
    // Methods
    public SwingListbox(
        Element self,
        XulComponent parent,
        XulDomContainer container,
        String tagName
    ) {}
    public Object getManagedObject() {}
    ...
    public void setCommand(final String command) {}
    // Fields
    public int counter = 0;
    // Method under Test
    public String getSeltype() {
        return selType;
    }}
}
// Test Class:
import static org.junit.Assert.assertEquals;
...
import org.pentaho.ui.xul.swing.SwingXulLoader;

public class SwingListboxTest  {
    // Fields
    Document doc = null;
    XulDomContainer container;
    XulListbox list;
    // Helper Methods
    @Before public void setUp() throws Exception {
      // Do not run on headless environment
      Assume.assumeTrue(!GraphicsEnvironment.isHeadless());
      container = new SwingXulLoader().loadXul("documents/listbox.xul");
      doc = container.getDocumentRoot();
      list = (XulListbox) doc.getElementById("listbox");
    }
    private static String toString(int[] is) {}
    // Unit Test Case
    @Test public void testSeltype() throws Exception {
      assertEquals("single", list.getSeltype());
    }
}
```

Class Name and Constructor **Priority 6**

Focal Class Methods **Priority 2**

Focal Class Fields **Priority 1**

MUT **Priority 6**

Test Class Imports **Priority 4**

Class Name **Priority 6**

Test Class Fields **Priority 5**

Test Class Methods **Priority 3**

Test Case **Priority 6**

Figure 5.1: Example of the formatting of focal class and test class context. color schema on the right indicates prioritization of components when not all context fits into the token limit. Priority 6 means highest priority, Priority 1 means lowest priority.

covered lines and total lines, as our main evaluation metric. Furthermore, we consider the *Correct* metric proposed by Tufano et al. [47] measuring the fraction of test cases that are executable and call the correct MUT. We furthermore report the test loss, BLEU and crystalBLEU scores as intrinsic metrics.

## 5.2 Research Questions

RQ1 **Training Methods: How effective is prefix tuning for test case generation?**

Prefix Tuning has been applied to a plethora of tasks in NLP and code intelligence, but there is not a clear consensus if it is worse than, equal to or better than full fine-tuning in terms of the final performance. To the best of our knowledge, no papers have applied prefix tuning as the main training method to the task of test case generation. Zlotchevski et al. [54] use prefix tuning for adapting to specific repositories, but the model was first fully fine-tuned on test case generation. We close that gap by comparing the performance of prefix tuning, full fine-tuning and zero-shot prompting on Tests4J.

RQ2 **Context Information: How does context information affect the performance when provided in the hard prompt?**

Tufano et al. [47] found that providing AthenaTest with focal class context reduces its loss by about 10%. However, they do not evaluate the performance impact with further, more important metrics such as coverage or BLEU scores. In contrast, we evaluate the effect of context using Tests4J, presenting both intrinsic and execution-based metrics. We always provide as much context as possible using the method described in subsection 5.1.1 and additionally consider the test class context.

RQ3 **Effectiveness of CAPT: Can CAPT effectively compress context information while retaining performance?**

Motivated by the importance of context information and the quadratic complexity of transformers, we proposed Context-Aware Prompt Tuning (CAPT) in chapter 4. In this RQ, we evaluate the effectiveness of CAPT. We first investigate the limits of how much information can be stored in a single soft token to get foundational insights about the mechanism. Afterward, we evaluate CAPT on Tests4J and compare the performance between not providing context, providing it via CAPT and providing it in the hard prompt.

RQ4 **Comparison with Evosuite: How do neural models compare to the state-of-the-art non-neural test case generator on Tests4J?**

Evosuite [19] is one of the most established tools for non-neural test case generation and typically achieves very high coverage scores. In a study [20], Evosuite was even found to outperform human developers in terms of coverage. While it excels at generating tests with high coverage, other quality properties like readability and maintainability lack behind human-written tests [22, 38, 23, 14]. In contrast, neural models are specifically trained to generate

tests that look similar to their training distribution, i.e., to human-written code. Consequently, they generate test cases that are much more readable for humans. For instance, 61% of human developers favored AthenaTest's test cases over Evosuite's in terms of readability, 10% favored Evosuite's and 29% found them equally readable [47]. In this work, we do not further evaluate readability, but focus on the performance metrics computed by Tests4J. The authors of AthenaTest found that it achieved comparable coverage to Evosuite on a single focal class. We argue that due to the high variance in coverage between repositories, an evaluation on such a small scale is not sufficient. We close that gap by evaluating neural models and Evosuite on Tests4J, which includes 484 focal classes from 11 repositories in the test set. To the best of our knowledge, we are the first to compare Evosuite to neural approaches at scale.

RQ5 **Evaluation Methodology: How important is large-scale coverage evaluation?**

Tests4J computes both intrinsic and execution-based metrics for test case generators. Among these, loss is by far the easiest to compute because it is available at training time and does not require execution. BLEU and crystalBLEU are available after sampling from the trained LM. Finally, the coverage and count metrics from Tests4J require both sampling from the LM and execution, which makes them the most time-consuming. To guide future research in the methodology of training and evaluating neural test case generators, we investigate the importance of large-scale coverage evaluation. In particular, we aim to find out if the cheaper to compute metrics mentioned above are good indicators of coverage and how important the scale is for a reliable evaluation.

## 5.3 Results

In this section, we present the results of the research questions introduced in section 5.2. Table A.21 summarizes the results of all evaluated setups. The detailed evaluation tables are presented at the respective RQs. We name every run according to their training algorithm, context information and the way the context is represented. More specifically, runs names start with the training algorithm (FT for fine-tuning, PT for prefix tuning and ZS for zero-shot), followed by the context in the hard prompt as Hard-All, Hard-TCl, Hard-FCl or Hard-None, and the context in the soft prompt as [PG]-All, [PG]-TCl or [PG]-FCl where [PG] is one of the prompt generator architectures presented in chapter 4.

| Name | Training Algorithm | Context | | Metrics | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Test Class (TCl) | Focal Class (FCl) | MiLC | Correct | Loss | crystalBLEU | BLEU |
| Evosuite | - | - | - | 37.78% | - | - | - | - |
| FT-Hard-All | Fine-tuning | Hard | Hard | 13.55% | 2.81% | 0.6743 | 0.0473 | 0.0808 |
| ZS-Hard-All | Zero-Shot | Hard | Hard | 3.07% | 0.29% | 1.0330 | 0.0237 | 0.0509 |
| PT-Hard-All | Prefix Tuning | Hard | Hard | 9.76% | 1.16% | 0.6825 | 0.0355 | 0.0677 |
| PT-Hard-TCl | Prefix Tuning | Hard | - | 5.48% | 0.82% | 0.6887 | 0.0378 | 0.0691 |
| PT-Hard-FCl | Prefix Tuning | - | Hard | 3.28% | 0.73% | 0.8091 | 0.0323 | 0.0647 |
| PT-Hard-None | Prefix Tuning | - | - | 1.79% | 0.12% | 0.8766 | 0.0283 | 0.0550 |
| PT-Hard-None-BERT-TCl-1 | Prefix Tuning | BERT ($l = 1$) | - | 1.79% | 0.19% | 0.8944 | 0.0257 | 0.0557 |
| PT-Hard-None-BERT-TCl-10 | Prefix Tuning | BERT ($l = 10$) | - | 1.82% | 0.19% | 0.8937 | 0.0285 | 0.0586 |
| PT-Hard-None-BERT_C-TCl-30 | Prefix Tuning | BERT_C ($l = 30$) | - | 2.03% | 0.23% | 0.9004 | 0.0304 | 0.0611 |
| PT-Hard-None-BERT_C-TCl-90 | Prefix Tuning | BERT_C ($l = 90$) | - | 1.92% | 0.28% | 0.9051 | 0.0328 | 0.0654 |
| PT-Hard-None-BERT_C-TCl-150 | Prefix Tuning | BERT_C ($l = 150$) | - | 1.89% | 0.23% | 1.1792 | 0.0306 | 0.0627 |
| PT-Hard-None-BERT_C-TCl-188 | Prefix Tuning | BERT_C ($l = 188$) | - | 1.79% | 0.38% | 1.1810 | 0.0312 | 0.0632 |
| PT-Hard-None-LSTM-TCl-1 | Prefix Tuning | LSTM ($l = 1$) | - | 1.49% | 0.18% | 0.8957 | 0.0281 | 0.0587 |
| PT-Hard-None-LSTM-TCl-10 | Prefix Tuning | LSTM ($l = 10$) | - | 1.81% | 0.22% | 0.8908 | 0.0305 | 0.0618 |
| PT-Hard-None-LSTM_C-TCl-30 | Prefix Tuning | LSTM_C ($l = 30$) | - | 1.55% | 0.24% | 0.8964 | 0.0322 | 0.0644 |
| PT-Hard-None-LSTM_C-TCl-90 | Prefix Tuning | LSTM_C ($l = 90$) | - | 2.03% | 0.27% | 0.8941 | 0.0295 | 0.0608 |
| PT-Hard-None-LSTM_C-TCl-150 | Prefix Tuning | LSTM_C ($l = 150$) | - | 1.77% | 0.22% | 0.8951 | 0.0289 | 0.0596 |
| PT-Hard-None-LSTM_C-TCl-188 | Prefix Tuning | LSTM_C ($l = 188$) | - | 1.68% | 0.19% | 1.1903 | 0.0292 | 0.0601 |
| PT-Hard-None-Sum-TCl-1 | Prefix Tuning | Sum ($l = 1$) | - | 1.85% | 0.22% | 0.8939 | 0.0272 | 0.0532 |
| PT-Hard-None-Sum-TCl-10 | Prefix Tuning | Sum ($l = 10$) | - | 1.95% | 0.32% | 0.8978 | 0.0291 | 0.0560 |
| PT-Hard-None-Sum_C-TCl-30 | Prefix Tuning | Sum_C ($l = 30$) | - | 2.08% | 0.23% | 0.8939 | 0.0292 | 0.0604 |
| PT-Hard-None-Sum_C-TCl-90 | Prefix Tuning | Sum_C ($l = 90$) | - | 1.76% | 0.26% | 0.8922 | 0.0296 | 0.0608 |
| PT-Hard-None-Sum_C-TCl-150 | Prefix Tuning | Sum_C ($l = 150$) | - | 1.58% | 0.32% | 0.8929 | 0.0292 | 0.0597 |
| PT-Hard-None-Sum_C-TCl-188 | Prefix Tuning | Sum_C ($l = 188$) | - | 1.98% | 0.24% | 0.8937 | 0.0296 | 0.0611 |

Table 5.1: Summary of the results.

### 5.3.1 RQ1: Training Methods

To compare the three training paradigms, we use the setup with all context information in the hard prompt. We find that full fine-tuning (Table 5.2) significantly outperforms prefix tuning (Table 5.3). For instance, the MiLC is almost 40% higher (13.55% and 9.76%) and the number of correct test cases is more than twice as high (625 and 274). Moreover, both the coverage and the fraction of correct test cases is higher on every project except one. The loss of full fine-tuning is better as well, but the difference is rather small ($\sim 1.2\%$) compared to the other metrics.

The performance of PolyCoder in a zero-shot setting (Table 5.4) is unsurprisingly much worse than both fine-tuning and prefix tuning. For three repositories, it failed to generate any correct test cases. The fraction of parsable test cases, however, is about the same as with full fine-tuning and prefix tuning. We argue that this is because the ability to generate syntactically correct code is mainly acquired during pre-training. We furthermore find that the fraction of compilable test cases is very similar between all training setups, and that the fraction of executable test cases is much higher in the zero-shot setting. We hypothesize that this is because without training, the model will generate a large fraction of meaningless test cases that, for instance, only declare a number of primitive variables without ever invoking any project functionality. Thereby, these test cases avoid the challenge of correctly using local APIs. We find support for this hypothesis by comparing the fraction of executable test cases that invoke the MUT ($\frac{Correct}{Executable}$) and the fraction of test cases that use at least one assert statement. Both values are significantly higher for prefix tuning than for zero-shot and significantly higher for full fine-tuning than for prefix tuning. We conclude that generating test cases that invoke the MUT and assert the program state is a skill acquired during fine-tuning and that full fine-tuning is more effective than prefix tuning in that regard.

In all models, the rate of compilable tests is rather low (12.24% to 13.07%) compared to the

FT-Hard-All loss: 0.6743 BLEU: 0.0808 crystalBLEU: 0.0473

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 6.91% | 6.43% | 5.58% | 5.79% | 1231 | 1050 (85.30%) | 157 (12.75%) | 56 (4.55%) | 11 (0.89%) |
| bazaarvoice-ostrich | 35.40% | 34.04% | 24.60% | 30.99% | 1271 | 1106 (87.02%) | 209 (16.44%) | 129 (10.15%) | 32 (2.52%) |
| adorsys-XS2A-Sandbox | 4.82% | 7.72% | 7.73% | 6.92% | 3196 | 2991 (93.59%) | 226 (7.07%) | 155 (4.85%) | 39 (1.22%) |
| flipkart-incubator-Lyrics | 6.95% | 17.88% | 13.49% | 17.64% | 1253 | 1161 (92.66%) | 78 (6.23%) | 47 (3.75%) | 14 (1.12%) |
| seedstack-seed | 15.57% | 25.47% | 11.92% | 21.60% | 934 | 808 (86.51%) | 139 (14.88%) | 62 (6.64%) | 42 (4.50%) |
| intuit-QuickBooks-V3-Java-SDK | 18.06% | 25.07% | 19.85% | 24.24% | 2012 | 1830 (90.95%) | 355 (17.64%) | 124 (6.16%) | 62 (3.08%) |
| gooddata-gooddata-java | 6.86% | 13.27% | 8.96% | 12.44% | 4065 | 3563 (87.65%) | 720 (17.71%) | 390 (9.59%) | 152 (3.74%) |
| shroffk-phoebus | 10.15% | 14.15% | 7.74% | 12.78% | 3373 | 2865 (84.94%) | 479 (14.20%) | 217 (6.43%) | 106 (3.14%) |
| messaginghub-pooled-jms | 10.68% | 8.28% | 18.77% | 7.05% | 1987 | 1925 (96.88%) | 125 (6.29%) | 62 (3.12%) | 47 (2.37%) |
| nhl-dflib | 12.94% | 16.26% | 12.75% | 13.28% | 1645 | 1507 (91.61%) | 185 (11.25%) | 89 (5.41%) | 51 (3.10%) |
| monarch-initiative-phenol | 40.49% | 38.52% | 39.64% | 38.22% | 1310 | 1203 (91.83%) | 238 (18.17%) | 99 (7.56%) | 69 (5.27%) |
| Total | 13.55% | 18.83% | 13.60% | 17.36% | 22277 | 20009 (89.82%) | 2911 (13.07%) | 1430 (6.42%) | 625 (2.81%) |

Table 5.2: Results for FT-Hard-All, the model including test class and focal class context trained with full fine-tuning.

PT-Hard-All loss: 0.6825 BLEU: 0.0677 crystalBLEU: 0.0355

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 4.07% | 4.69% | 4.65% | 4.04% | 1346 | 1201 (89.23%) | 174 (12.93%) | 101 (7.50%) | 15 (1.11%) |
| bazaarvoice-ostrich | 26.55% | 25.09% | 16.67% | 24.65% | 1358 | 1253 (92.27%) | 327 (24.08%) | 219 (16.13%) | 19 (1.40%) |
| adorsys-XS2A-Sandbox | 1.66% | 4.58% | 3.86% | 4.62% | 3203 | 2833 (88.45%) | 190 (5.93%) | 127 (3.97%) | 21 (0.66%) |
| flipkart-incubator-Lyrics | 5.02% | 8.03% | 7.94% | 7.50% | 1190 | 1124 (94.45%) | 223 (18.74%) | 136 (11.43%) | 4 (0.34%) |
| seedstack-seed | 9.56% | 11.71% | 7.77% | 11.17% | 957 | 811 (84.74%) | 114 (11.91%) | 61 (6.37%) | 26 (2.72%) |
| intuit-QuickBooks-V3-Java-SDK | 17.29% | 20.85% | 17.62% | 19.91% | 2127 | 1914 (89.99%) | 480 (22.57%) | 254 (11.94%) | 53 (2.49%) |
| gooddata-gooddata-java | 3.51% | 8.16% | 5.07% | 7.64% | 4696 | 4261 (90.74%) | 567 (12.07%) | 349 (7.43%) | 40 (0.85%) |
| shroffk-phoebus | 5.43% | 6.23% | 4.19% | 5.63% | 3450 | 2834 (82.14%) | 341 (9.88%) | 203 (5.88%) | 42 (1.22%) |
| messaginghub-pooled-jms | 19.90% | 10.62% | 22.22% | 8.53% | 2216 | 2133 (96.25%) | 120 (5.42%) | 62 (2.80%) | 13 (0.59%) |
| nhl-dflib | 5.39% | 7.96% | 4.78% | 7.00% | 1658 | 1403 (84.62%) | 156 (9.41%) | 59 (3.56%) | 16 (0.97%) |
| monarch-initiative-phenol | 20.38% | 21.36% | 19.53% | 21.30% | 1405 | 1177 (83.77%) | 198 (14.09%) | 64 (4.56%) | 25 (1.78%) |
| Total | 9.76% | 11.75% | 9.37% | 11.09% | 23606 | 20944 (88.72%) | 2890 (12.24%) | 1635 (6.93%) | 274 (1.16%) |

Table 5.3: Results for PT-Hard-All, the model including test class and focal class context trained with prefix tuning.

numbers reported in the AthenaTest paper ($\sim 50\%$). Upon inspection, we find that the majority ($70 - 90\%$) of compilation errors are `cannot find symbol`. We argue that this is likely due to the different evaluation setup, where generated test cases are inserted into fixed test classes instead of having a test class specifically constructed for them. That is, our models need to maintain consistency with the namespace in the test class. We find that this ability is also learned during fine-tuning, indicated by a reduced fraction of symbol errors among all compilation errors (80% for zero-shot, 74% for prefix tuning and 70% for full fine-tuning).

**RQ1 Main Takeaways**

Prefix Tuning (MiLC 9.76%, Correct 1.16%) lies between zero-shot (MiLC 3.07%, Correct 0.29%) and full fine-tuning (MiLC 13.55%, Correct 2.81%). Fine-tuning reduces the number of meaningless test cases that don't invoke the MUT or don't assert anything, and improves consistency with the local namespace. In both aspects, full fine-tuning is more effective than prefix tuning.

ZS-Hard-All loss: 1.0330 BLEU: 0.0509 crystalBLEU: 0.0237

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 3.66% | 2.94% | 4.19% | 2.28% | 1357 | 1248 (91.97%) | 209 (15.40%) | 160 (11.79%) | 1 (0.07%) |
| bazaarvoice-ostrich | 21.24% | 17.26% | 11.11% | 16.20% | 1481 | 1401 (94.60%) | 220 (14.85%) | 184 (12.42%) | 6 (0.41%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 3521 | 3115 (88.47%) | 456 (12.95%) | 393 (11.16%) | 0 (0.00%) |
| flipkart-incubator-Lyrics | 0.00% | 0.00% | 0.00% | 0.00% | 1192 | 1078 (90.44%) | 186 (15.60%) | 171 (14.35%) | 0 (0.00%) |
| seedstack-seed | 4.37% | 5.60% | 5.18% | 4.47% | 1105 | 1020 (92.31%) | 112 (10.14%) | 58 (5.25%) | 7 (0.63%) |
| intuit-QuickBooks-V3-Java-SDK | 1.28% | 7.54% | 2.73% | 6.89% | 2494 | 2185 (87.61%) | 476 (19.09%) | 375 (15.04%) | 26 (1.04%) |
| gooddata-gooddata-java | 0.61% | 1.90% | 1.19% | 1.90% | 5109 | 4580 (89.65%) | 703 (13.76%) | 555 (10.86%) | 2 (0.04%) |
| shroffk-phoebus | 2.60% | 1.89% | 1.65% | 1.52% | 3671 | 2995 (81.59%) | 356 (9.70%) | 232 (6.32%) | 14 (0.38%) |
| messaginghub-pooled-jms | 5.83% | 4.95% | 4.60% | 4.09% | 2456 | 2134 (86.89%) | 126 (5.13%) | 73 (2.97%) | 0 (0.00%) |
| nhl-dflib | 4.58% | 7.26% | 4.78% | 6.00% | 1663 | 1240 (74.56%) | 180 (10.82%) | 111 (6.67%) | 12 (0.72%) |
| monarch-initiative-phenol | 4.62% | 6.67% | 4.14% | 6.88% | 1482 | 1316 (88.80%) | 185 (12.48%) | 96 (6.48%) | 6 (0.40%) |
| Total | 3.07% | 5.09% | 2.99% | 4.57% | 25531 | 22312 (87.39%) | 3209 (12.57%) | 2408 (9.43%) | 74 (0.29%) |

Table 5.4: Results for ZS-Hard-All, the model including test class and focal class context without fine-tuning.

### 5.3.2 RQ2: Context Information


Figure 5.2: Relative increases achieved by focal class and test class context.

We find that, as hypothesized in the introduction, test case generation is indeed a task that profits a lot from contextual information. Compared to the model without any additional context (PT-Hard-None, Table 5.7), the model using both focal class and test class context (PT-Hard-All, Table 5.3) achieves over a 4x increase in MiLC from 1.79% to 9.76% and about a 10x increase in the number of correct test cases. We furthermore find that both focal class context (PT-Hard-FCl, Table 5.6) and test class context (PT-Hard-TCl, Table 5.5) are crucial, with 3.28% and 5.48% MiLC respectively. The test class context results in the larger relative improvement of 206% compared to 83% relative improvement by the focal class. These relative improvements translate almost equivalently to the setups where the respective other context is already given, as illustrated in Figure 5.2. That is, the MiLC of PT-Hard-All is 197% higher than that of PT-Hard-FCl and 78% higher than that of PT-Hard-TCl. This indicates that the two context types are highly complementary. Lastly, we find that context information can lead to an even larger improvement than fine-tuning. For instance, the zero-shot model with all context (ZS-Hard-All) performs significantly better than the prefix tuning model without context (PT-Hard-None).

**RQ2 Main Takeaways**

The results show that adding context information to the model increases the MiLC by more than 4x and that context is even more important than fine-tuning. Test class context is more effective than focal class context individually, but they are highly complementary.

PT-Hard-TCl loss: 0.6887 BLEU: 0.0691 crystalBLEU: 0.0378

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1360 | 1188 (87.35%) | 134 (9.85%) | 94 (6.91%) | 3 (0.22%) |
| bazaarvoice-ostrich | 30.97% | 28.83% | 19.84% | 23.94% | 1415 | 1306 (92.30%) | 186 (13.14%) | 128 (9.05%) | 18 (1.27%) |
| adorsys-XS2A-Sandbox | 1.51% | 2.66% | 2.90% | 2.31% | 3301 | 2853 (86.43%) | 283 (8.57%) | 203 (6.15%) | 6 (0.18%) |
| flipkart-incubator-Lyrics | 0.77% | 3.33% | 1.59% | 3.33% | 1519 | 1067 (70.24%) | 143 (9.41%) | 100 (6.58%) | 2 (0.13%) |
| seedstack-seed | 8.20% | 10.67% | 6.22% | 8.19% | 1111 | 983 (88.48%) | 145 (13.05%) | 99 (8.91%) | 15 (1.35%) |
| intuit-QuickBooks-V3-Java-SDK | 3.07% | 8.13% | 4.71% | 7.90% | 2321 | 2123 (91.47%) | 528 (22.75%) | 281 (12.11%) | 40 (1.72%) |
| gooddata-gooddata-java | 1.98% | 3.79% | 2.39% | 3.79% | 4774 | 4413 (92.44%) | 640 (13.41%) | 507 (10.62%) | 32 (0.67%) |
| shroffk-phoebus | 2.12% | 5.63% | 1.78% | 4.77% | 3739 | 2812 (75.21%) | 142 (3.80%) | 101 (2.70%) | 27 (0.72%) |
| messaginghub-pooled-jms | 20.63% | 10.73% | 21.84% | 8.77% | 2293 | 2052 (89.49%) | 147 (6.41%) | 112 (4.88%) | 12 (0.52%) |
| nhl-dflib | 1.89% | 4.00% | 1.59% | 2.67% | 1821 | 1240 (68.09%) | 57 (3.13%) | 26 (1.43%) | 8 (0.44%) |
| monarch-initiative-phenol | 13.32% | 16.41% | 13.02% | 15.22% | 1363 | 1227 (90.02%) | 201 (14.75%) | 92 (6.75%) | 43 (3.15%) |
| Total | 5.48% | 8.70% | 5.76% | 7.48% | 25017 | 21264 (85.00%) | 2606 (10.42%) | 1743 (6.97%) | 206 (0.82%) |

Table 5.5: Results for PT-Hard-TCl, the model including test class context trained with prefix tuning.

PT-Hard-FCl loss: 0.8091 BLEU: 0.0647 crystalBLEU: 0.0323

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 0.00% | 0.00% | 0.00% | 0.00% | 1216 | 994 (81.74%) | 52 (4.28%) | 46 (3.78%) | 1 (0.08%) |
| bazaarvoice-ostrich | 10.18% | 13.15% | 8.73% | 12.68% | 1452 | 1176 (80.99%) | 64 (4.41%) | 39 (2.69%) | 9 (0.62%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 3014 | 2753 (91.34%) | 405 (13.44%) | 367 (12.18%) | 7 (0.23%) |
| flipkart-incubator-Lyrics | 0.00% | 0.00% | 0.00% | 0.00% | 1288 | 1170 (90.84%) | 42 (3.26%) | 31 (2.41%) | 0 (0.00%) |
| seedstack-seed | 1.64% | 1.28% | 1.55% | 1.06% | 974 | 900 (92.40%) | 14 (1.44%) | 9 (0.92%) | 2 (0.21%) |
| intuit-QuickBooks-V3-Java-SDK | 1.36% | 5.59% | 2.73% | 4.60% | 2296 | 2092 (91.11%) | 129 (5.62%) | 97 (4.22%) | 12 (0.52%) |
| gooddata-gooddata-java | 2.44% | 3.79% | 2.39% | 3.79% | 4487 | 4234 (94.36%) | 227 (5.06%) | 221 (4.93%) | 21 (0.47%) |
| shroffk-phoebus | 3.30% | 6.35% | 2.41% | 6.20% | 3407 | 3015 (88.49%) | 329 (9.66%) | 235 (6.90%) | 54 (1.58%) |
| messaginghub-pooled-jms | 10.44% | 5.73% | 16.48% | 4.37% | 2230 | 2151 (96.46%) | 82 (3.68%) | 55 (2.47%) | 32 (1.43%) |
| nhl-dflib | 3.77% | 7.56% | 3.59% | 6.00% | 1803 | 1328 (73.66%) | 135 (7.49%) | 51 (2.83%) | 31 (1.72%) |
| monarch-initiative-phenol | 12.23% | 11.40% | 9.47% | 12.32% | 1497 | 1235 (82.50%) | 129 (8.62%) | 104 (6.95%) | 3 (0.20%) |
| Total | 3.28% | 4.99% | 3.90% | 4.64% | 23664 | 21048 (88.95%) | 1608 (6.80%) | 1255 (5.30%) | 172 (0.73%) |

Table 5.6: Results for PT-Hard-FCl, the model including focal class context trained with prefix tuning.

PT-Hard-None loss: 0.8766 BLEU: 0.0550 crystalBLEU: 0.0283

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1209 | 984 (81.39%) | 79 (6.53%) | 78 (6.45%) | 0 (0.00%) |
| bazaarvoice-ostrich | 1.33% | 2.82% | 1.59% | 2.82% | 1395 | 1188 (85.16%) | 43 (3.08%) | 32 (2.29%) | 4 (0.29%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2946 | 2695 (91.48%) | 433 (14.70%) | 384 (13.03%) | 2 (0.07%) |
| flipkart-incubator-Lyrics | 0.00% | 0.00% | 0.00% | 0.00% | 1377 | 976 (70.88%) | 41 (2.98%) | 20 (1.45%) | 0 (0.00%) |
| seedstack-seed | 1.09% | 2.13% | 0.52% | 2.13% | 902 | 771 (85.48%) | 32 (3.55%) | 30 (3.33%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2342 | 2038 (87.02%) | 85 (3.63%) | 79 (3.37%) | 2 (0.09%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2838 | 2688 (94.71%) | 128 (4.51%) | 122 (4.30%) | 1 (0.04%) |
| shroffk-phoebus | 0.79% | 1.97% | 0.89% | 1.73% | 3348 | 2593 (77.45%) | 144 (4.30%) | 85 (2.54%) | 9 (0.27%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1641 | 1563 (95.25%) | 33 (2.01%) | 28 (1.71%) | 0 (0.00%) |
| nhl-dflib | 2.96% | 0.98% | 2.79% | 0.78% | 1570 | 1284 (81.78%) | 34 (2.17%) | 22 (1.40%) | 0 (0.00%) |
| monarch-initiative-phenol | 15.49% | 12.09% | 13.02% | 12.97% | 1408 | 1119 (79.47%) | 63 (4.47%) | 46 (3.27%) | 8 (0.57%) |
| Total | 1.79% | 2.15% | 1.69% | 2.09% | 20976 | 17899 (85.33%) | 1115 (5.32%) | 926 (4.41%) | 26 (0.12%) |

Table 5.7: Results for PT-Hard-None, the model without context information trained with prefix tuning.

### 5.3.3 RQ3: Effectiveness of CAPT

In chapter 4, we presented Context-Aware Prompt Tuning (CAPT), which compresses context information before passing it to the language model with the goal of mitigating the transformer's quadratic complexity. To validate the basic mechanism, we first perform a preliminary study in which the target sequence is passed to the prompt generator. That is, the model is essentially acting as an autoencoder, reconstructing the input from a dense representation. If the model is able to do that, it means that the information was effectively compressed into a representation that guides the model in generating the correct target. We perform this task with random data, i.e., we construct random token sequences $[x_0, ..., x_{n_{\text{tokens}}-1}]$, let the prompt generator compress it into a single soft token and train the LM to reconstruct the sequence in the output. To test the limits of compression, we vary the random token sequences in two dimensions. $n_{\text{tokens}}$ is the number of tokens in the sequence, and $n_{\text{choices}}$ is the number of values every token $x_i$ can take. We denote $T_{\text{train}}(n)$ as the set of the $n$ most frequent tokens in the train set. Then every random token is sampled uniformly from that set, i.e., $x_i \sim U(T_{\text{train}}(n_{\text{choices}}))$. With increasing values in both dimensions, we expect the task to become harder because the information content is increased. We evaluate the performance of different PGs as multi-label classification task on the accuracy. In particular, we evaluate the PGs listed in the legend of Figures 5.3 and 5.4. Note that they are not all completely equivalent to those presented in chapter 4 because this experiment was used to guide the design of these final PGs. CodeBERT(frozen, sum) is the same as the PG BERT, except that we freeze the CodeBERT model and use the sum of all token embeddings as aggregation. LSTM(2 layers) is the same as the PG LSTM, except that we use 2 layers here. LSTM(1 layer), CodeBERT(train, CLS) and Sum are used as described in chapter 4. For every prompt generator, we test a variant with shallow soft tokens that are only injected at the input layer (Figure 5.4) and a variant with deep soft tokens that span all layers (Figure 5.3). We use $n_{\text{tokens}} \in \{1, 10, 100\}$ and $n_{\text{choices}} \in \{2, 10, 100, 1000\}$ for both deep and shallow soft tokens, resulting in 120 training runs.

The results indicate that the information flow is not optimal. Even with a single token, where no compression is happening, some PGs fail to correctly steer the LM. This is particularly the case for

the PGs based on CodeBERT. Furthermore, the performance drops quickly when the complexity (i.e. either one of the parameters) is increased. We follow that the amount of information that can be effectively encoded into a single soft token is very limited. Therefore, we conclude that effectively scaling the number of soft tokens and alleviating the information bottleneck is crucial for real context, motivating the chunked PG variants introduced in chapter 4. We find that the trends are very similar between shallow soft tokens that are only injected at the input layer and deep ones that span all layers. The performance seems to drop slightly quicker with shallow soft tokens. Because of that and because of missing support in huggingface for shallow soft tokens during generation, we focus on deep soft tokens in all further experiments, as discussed in chapter 4.

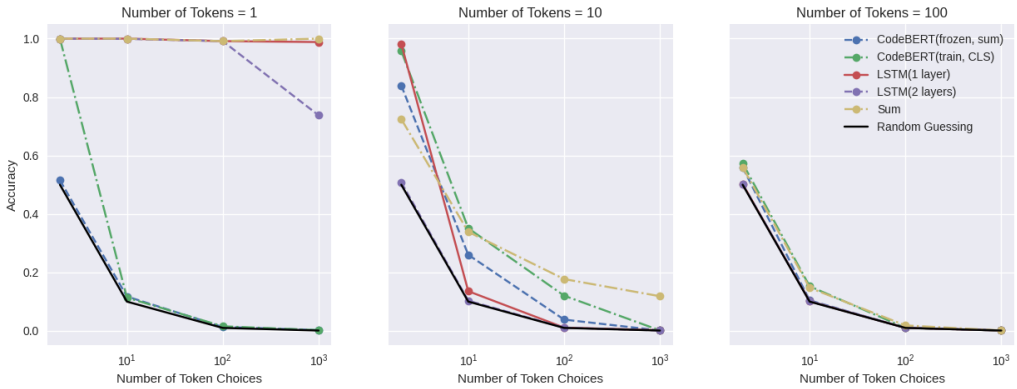



Figure 5.3: Results of the preliminary study on the limits of compression with soft prompts that span all layers.

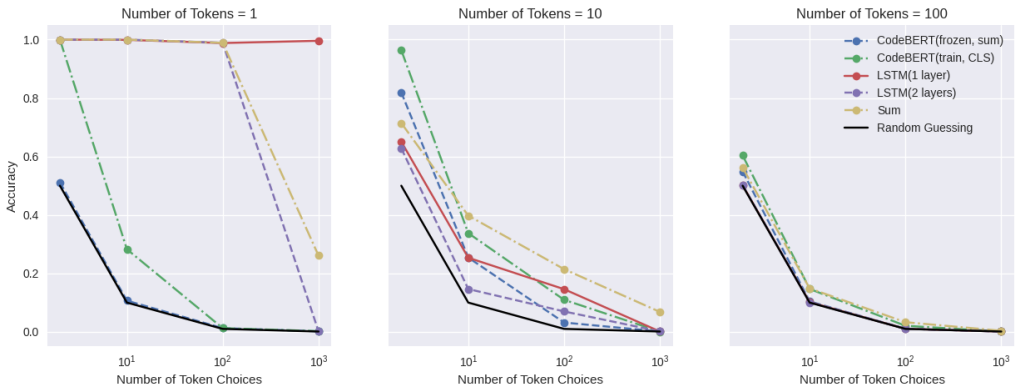



Figure 5.4: Results of the preliminary study on the limits of compression with soft prompts that are injected only at the input layer.

Next, we apply CAPT to the actual task of test case generation with compressed context. We see the hard prompt variant as a target because it allows the LM to attend to all context tokens, whereas in CAPT, the LM can only attend to the compressed representation that likely suffers from some degree of information loss. We expect the model that uses CAPT to lie somewhere between a model that does not have the context information at all and a model that has the context information in the hard prompt. In particular, we use PT-Hard-None (Table 5.7) as the baseline and PT-Hard-TCl (Table 5.3) as the target. For the sake of simplicity, we only consider this setting and do not experiment with focal class context at this point. We leave experimentation with other context types to future work. In the following, we denote $l$ as the number of soft tokens

used to encode the test class. We evaluate the three non-chunked variants with $l \in \{1, 10\}$ and the chunked variants with $l \in \{30, 90, 150, 188\}$. 188 is the maximum number because 198 tokens were reserved for soft tokens and 10 tokens are used for prefix tuning. The mean number of tokens in the test class context is 416, resulting in a mean compression rate of almost 1:2 with 188 soft tokens.

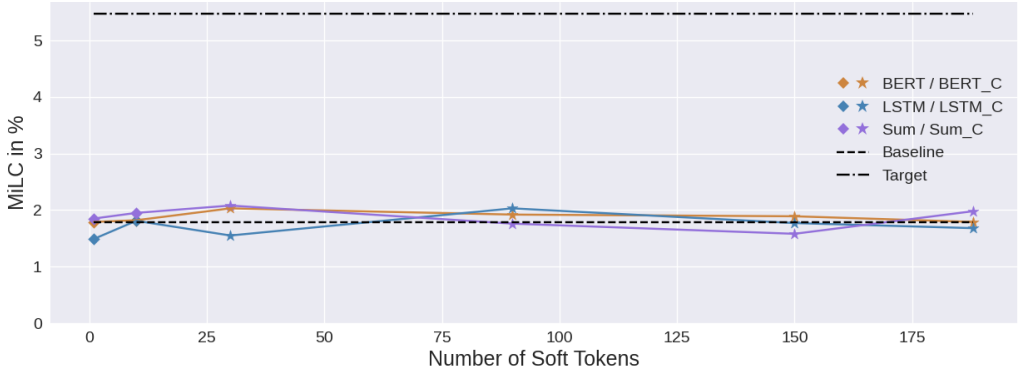



Figure 5.5: Results of CAPT on Tests4J with respect to the number of soft tokens. The baseline is PT-Hard-None and the target is PT-Hard-TCl.

Figure 5.5 shows the results of this experiment. The full results tables can be found in the appendix in section A.3. We find that the coverage achieved with compressed context is not substantially higher than without the context, and far from the coverage achieved with the context in the hard prompt. In fact, some of the configuration actually lead to lower coverage than the baseline. Furthermore, all CAPT configurations lead to a higher loss than the baseline. On the other hand, all CAPT configurations lead to a higher relative and absolute number of correct test cases, with an increase of up to 3x (0.12% → 0.38%). However, overall, the differences between CAPT configurations and the baseline are rather small compared to the target. We argue that both positive and negative differences in all metrics compared to the baseline could be due to the stochasticity of the generation process. The results furthermore suggest that there is no direct relationship between the number of soft tokens and the MiLC, indicated by weak Pearson correlations and p-values $\gg 0.05$ for all PGs. This is opposed to our previous intuition that the information bottleneck is the main limitation of the setup, and suggests that there are more fundamental problems with the mechanism. We leave further investigations about these problems to future work.

**RQ3 Main Takeaways**

CAPT does not yield results that are significantly better than the baseline. There is also no direct relationship between the number of soft tokens and the performance. This indicates that the information bottleneck is not the only limiting factor, and that there might be fundamental problems with the mechanism of aggregation and injection.

### 5.3.4 RQ4: Comparison with Evosuite

To generate test cases with Evosuite, we use 4 cores and 10GB of memory per repository. We specify the target classes and use the defaults for all other settings. That is, Evosuite has 2 minutes of runtime per class. Table 5.8 presents the results of Evosuite on Tests4J. Since Evosuite does not provide the information which MUT of a focal class a test case is targeting, we do not report intrinsic metrics and the *Correct* metric. For 4 out of the 11 repositories, Evosuite failed to generate test cases:

- In `monarch-initiative-phenol`, Evosuite fails due to a version conflict of the `jgrapht` library. Since the library introduced breaking changes to the API, executing Evosuite was not possible without porting Evosuite or the project to the respective other version. The same applies to `eclipse-winery` from the validation set.
- For `nhl-dflib`, Evosuite produced many tests that caused the Java VM to crash with out-of-memory errors. This problem persisted even after increasing the available memory to 50GB. Since the tests caused the Java VM to crash, filtering out the problematic test cases all at once was not possible. After manually removing numerous test cases and re-running the test execution every time, we decided to skip this repository.
- For `gooddata-gooddata-java` and `intuit-QuickBooks-V3-Java-SDK`, Evosuite's generation process fails silently and only leaves the error messages shown below in its internal logs. Since we could not identify the root cause and fix the error, we decided to remove these repositories from the evaluation as well.

```
[MASTER] 19:33:58.118 [main] ERROR SearchStatistics - No obtained value for output variable:
↪ LineCoverage
[MASTER] 19:33:58.219 [main] ERROR TestGeneration - failed to write statistics data
```

To maintain a fair comparison, we ignore these repositories when computing averages. Still, we note that compatibility is one drawback of Evosuite because it needs to integrate with the project and interference is possible. Since LMs view projects only as sequences of tokens, such interference does not happen with LMs.

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 30.28% | 41.71% | 25.12% | 40.70% | 234 | 234 (100.00%) | 234 (100.00%) | 231 (98.72%) | - |
| bazaarvoice-ostrich | 36.73% | 44.84% | 40.48% | 44.89% | 201 | 201 (100.00%) | 201 (100.00%) | 195 (97.01%) | - |
| adorsys-XS2A-Sandbox | 35.99% | 45.96% | 47.34% | 51.30% | 316 | 316 (100.00%) | 316 (100.00%) | 310 (98.10%) | - |
| flipkart-incubator-Lyrics | 18.53% | 30.02% | 36.51% | 34.38% | 212 | 212 (100.00%) | 212 (100.00%) | 210 (99.06%) | - |
| seedstack-seed | 34.70% | 52.85% | 34.72% | 49.32% | 183 | 183 (100.00%) | 183 (100.00%) | 182 (99.45%) | - |
| intuit-QuickBooks-V3-Java-SDK | - | - | - | - | - | - | - | - | - |
| gooddata-gooddata-java | - | - | - | - | - | - | - | - | - |
| shroffk-phoebus | 45.24% | 46.02% | 46.70% | 46.56% | 933 | 933 (100.00%) | 933 (100.00%) | 895 (95.93%) | - |
| messaginghub-pooled-jms | 41.99% | 50.09% | 60.54% | 54.21% | 441 | 441 (100.00%) | 441 (100.00%) | 437 (99.09%) | - |
| nhl-dflib | - | - | - | - | - | - | - | - | - |
| monarch-initiative-phenol | - | - | - | - | - | - | - | - | - |
| Total | 37.78% | 23.96% | 43.95% | 24.72% | 2520 | 2520 (100.00%) | 2520 (100.00%) | 2460 (97.62%) | - |

Table 5.8: Results for Evosuite

We compare these results to our best neural model, FT-Hard-All (Table 5.2). To enable a more direct comparison of aggregated metrics, we present the results of FT-Hard-All again in Table 5.9.

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 6.91% | 6.43% | 5.58% | 5.79% | 1231 | 1050 (85.30%) | 157 (12.75%) | 56 (4.55%) | 11 (0.89%) |
| bazaarvoice-ostrich | 35.40% | 34.04% | 24.60% | 30.99% | 1271 | 1106 (87.02%) | 209 (16.44%) | 129 (10.15%) | 32 (2.52%) |
| adorsys-XS2A-Sandbox | 4.82% | 7.72% | 7.73% | 6.92% | 3196 | 2991 (93.59%) | 226 (7.07%) | 155 (4.85%) | 39 (1.22%) |
| flipkart-incubator-Lyrics | 6.95% | 17.88% | 13.49% | 17.64% | 1253 | 1161 (92.66%) | 78 (6.23%) | 47 (3.75%) | 14 (1.12%) |
| seedstack-seed | 15.57% | 25.47% | 11.92% | 21.60% | 934 | 808 (86.51%) | 139 (14.88%) | 62 (6.64%) | 42 (4.50%) |
| intuit-QuickBooks-V3-Java-SDK | - | - | - | - | - | - | - | - | - |
| gooddata-gooddata-java | - | - | - | - | - | - | - | - | - |
| shroffk-phoebus | 10.15% | 14.15% | 7.74% | 12.78% | 3373 | 2865 (84.94%) | 479 (14.20%) | 217 (6.43%) | 106 (3.14%) |
| messaginghub-pooled-jms | 10.68% | 8.28% | 18.77% | 7.05% | 1987 | 1925 (96.88%) | 125 (6.29%) | 62 (3.12%) | 47 (2.37%) |
| nhl-dflib | - | - | - | - | - | - | - | - | - |
| monarch-initiative-phenol | - | - | - | - | - | - | - | - | - |
| Total | 10.68% | 10.36% | 10.91% | 9.34% | 13245 | 11906 (89.89%) | 1413 (10.67%) | 728 (5.50%) | 291 (2.20%) |

Table 5.9: Results for FT-Hard-All, without those repositories that Evosuite failed to run.

while ignoring the projects mentioned above. Evosuite outperforms this neural model in all coverage metrics. In particular, it covers more than 3 times as many lines. It is furthermore more consistent, indicated by the lower standard deviation of MiLC between repositories of 8.02, compared to 11.36 in FT-Hard-All. On the other hand, Evosuite also generated more than 3 times as many executable test cases. That is, on average, Evosuite covers about the same number of unique lines per test case (0.56) as FT-Hard-All (0.54).

**RQ4 Main Takeaways**

Evosuite covers almost 3 times as many lines and is more consistent across repositories than our best neural model. However, this 3 times increase is achieved by 3 times as many test cases, i.e., individual test cases generated by Evosuite and FT-Hard-All are similarly effective in terms of coverage. Evosuite failed to generate test cases for 4 out of the 11 projects.

### 5.3.5 RQ5: Evaluation Methodology

We first perform a correlation analysis of all metrics from all neural models, including full fine-tuning, prefix tuning and CAPT, in order to guide the methodology of future research. In particular, we aim to answer the question which metrics should be used for reliable evaluation in the process of model tuning and model selection. The results are displayed in Figure 5.6. We group the metrics according to the time when they are available. Training is denoted by T and includes only the loss. The generation is denoted by G and includes all metrics that are available after sampling from the LM, i.e., BLEU and crystalBLEU. The execution is denoted by E and includes all execution-based metrics. Unsurprisingly, the coverage metrics are all highly correlated. Among the count metrics, the fraction of compilable and the fraction of correct test cases correlate the strongest with the coverage metrics. This is in line with our previous insights that the parsability rate is very similar for all models and that the executable test cases include a significant amount of trivial test cases that don't invoke any project functionality. Furthermore, the results suggest that the loss might not be a very good indicator of task performance, given the rather low correlation to all other metrics ($<= 0.65$). Both BLEU scores correlate much stronger with the coverage and the rate of correct test cases. BLEU correlates slightly stronger than crystalBLEU with most execution-based

metrics, suggesting that BLEU is the better indicator of task performance in terms of execution-based metrics. For one, this questions the common practice to perform large-scale hyperparameter tuning and model selection only based on the loss. BLEU scores are much better indicators than loss, but we argue that measuring coverage directly is the most reliable option. Furthermore, coverage is much more interpretable and takes only about 15 minutes per model to obtain with Tests4J on a sufficiently capable machine. The results furthermore raise the question whether regular cross entropy loss is a good choice as an optimization target. We argue that additional loss terms could, for instance, provide a stronger signal about the consistency with the local namespace and enforce calling the MUT.

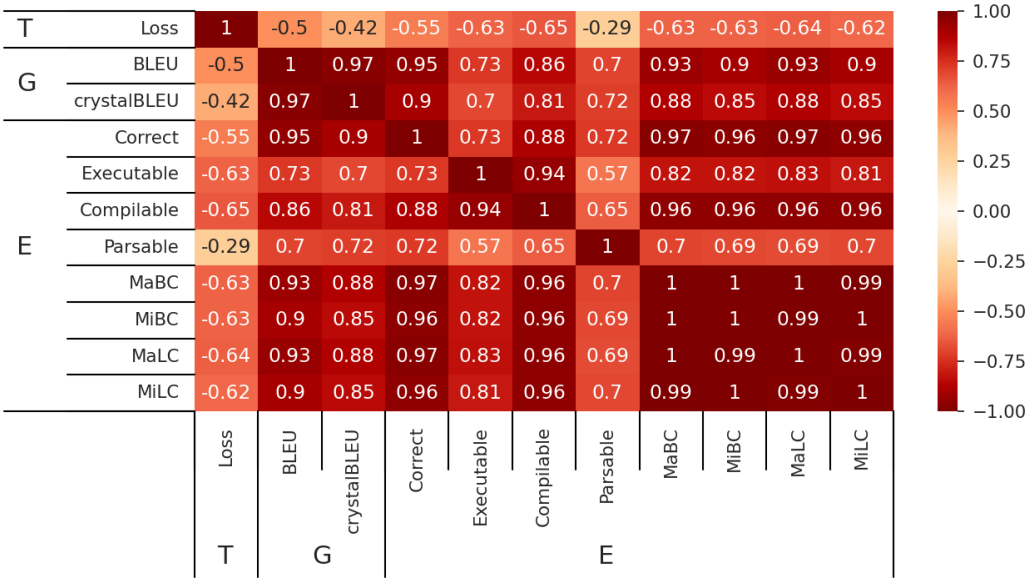



Figure 5.6: Heatmap of the Pearson correlations between metrics. E = Execution, G = Generation, T = Training.

Lastly, we note that there is a high variance in the coverage scores per project. For instance, FT-Hard-All achieves 40.49% MiLC on `monarch-initiative-phenol` but only 4.82% MiLC on `adorsys-XS2A-Sandbox` and even the zero-shot model with an aggregated MiLC of 3.07% achieves 21.21% MiLC on `bazaarvoice-ostrich`. We find that this variance is not just due to the stochasticity of the training and generation process. There is a clear trend that if a model performs well on a project compared to its average performance, it is likely that other models also perform well compared to their average performances. Thus, we follow that there are easier and more difficult projects. There are many potential reasons for this varying difficulty. For instance, overall complex constructor and MUT signatures could lead to a higher difficulty. We conclude that a large number of projects is essential for a reliable estimate of downstream performance.

**RQ5 Main Takeaways**

A correlation analysis among all metrics suggests that loss is a poor indicator of coverage. BLEU scores are much stronger indicators, but we argue that measuring coverage directly is the most reliable and interpretable option, and that it presents a low overhead to execute test cases with Tests4J after they were generated. We furthermore find that there is a large variance in the coverage between projects, and thus advocate for large-scale evaluations.

# 6 Discussion and Future Work

## 6.1 Tests4J Benchmark

During the dataset creation described in chapter 3, we started with a set of candidate repositories and filtered them according to several criteria. These criteria were defined with the execution-based evaluation in mind. For instance, repositories that failed to compile without external dependencies like databases were excluded. However, one might argue that some of the filters are not relevant during training, when no execution is required. We decided not to differentiate between training and evaluation in the process to avoid the risk of diverging distributions between the two stages. Future work should investigate this risk. If the evaluation setup is still i.i.d., introducing a differentiation between training and evaluation would be a simple way to further increase the benchmark scale significantly. In particular, future work could use all repositories that are excluded for reasons only relevant for execution as train set and split the remaining 60 repositories into validation and test, which would result in a size increase of about 10x for the training set and about 3x for the evaluation set. We reduce the barriers to such potential future work by saving all excluded repositories, their mapped test cases and the reason for exclusion. Besides increasing the dataset size, future work could also work on improving the data quality. In particular, we find that the benchmark contains a significant amount of very simple MUTs. For instance, about 20% of MUTs are getter and setter methods and the average cyclomatic complexity is 2.29. By filtering out such simple MUTs based on criteria like cyclomatic complexity might result in higher quality training data and a more challenging benchmark.

In the coverage pipeline (section 3.4), we insert generated test cases into the test class where the ground truth test case for the same MUT was located originally. We decided on this approach because it is an easy and effective way to get a test class that likely contains the most important imports and setup code. However, this approach also has its drawbacks. For one, it is not applicable in the scenario of a user generating test cases without having a test class scaffolding. But more importantly, we found in our experiments that the fraction of compilable test cases is only about 13%, while the majority of compilation errors are symbol errors. We argue that many of these symbol errors could be fixed by providing test classes that are not static, but specifically adjusted for a given test case. Future work might test approaches that either construct a test class from scratch or adjust the existing test class by adding imports and variable declarations.

Throughout this thesis, we considered coverage as the main metric to evaluate test cases. While coverage is one key quality attribute of a test case, correct assertions are also essential to detect regressions. Mutation scores might be a viable option to approximate the correctness of assertions.

Future work could integrate such additional metrics, for instance with PITest[1].

## 6.2 Modelling

In RQ1, we found that context information is essential for test case generation. It increases line coverage by over 4x and is even more effective than fine-tuning when compared in isolation. Furthermore, we found that focal class context and test class context are highly complementary, leading to roughly the same respective relative improvements in combination as in isolation. Therefore, we see adding even more context as a promising direction for future research. There are many options, for instance usage examples from documentation, snippets from methods that call the MUT, docstrings or few-shot examples from the same repository. However, additional context requires additional tokens, further motivating the usage of methods that extend the maximum sequence length.

In RQ3, we found that CAPT was not successful in effectively compressing and injecting context information. The experiments did not reveal a clear reason why, but we have a two possible, albeit untested hypotheses. First, a possible reason for the poor performance could be that there is too much responsibility placed on the PG. In our experiments, we found that alleviating the information bottleneck did not improve the results. This might indicate that the compressed context contains the correct information, or at least parts of it, but the LM cannot leverage it. In the current setup, the lightweight prompt generator has to produce an embedding that is meaningful in the LMs embedding space and effective in steering it. If the PGs are not complex and capable enough to do that, perhaps the LM weights should be adjusted as well, in order to learn to better leverage the compressed context. That is, future work should investigate the effect of using full fine-tuning in combination with CAPT. Second, the poor performance could also be due to the positioning of the compressed context. We inject compressed context via the `past_key_values` keyword of the Huggingface implementation of PolyCoder. Thereby, compressed context is inserted at the very beginning of the sequence. In RQ1, we found that even zero-shot, PolyCoder can generate some correct test cases when given good context. We believe this is only possible because the context is formatted in a coherent way (i.e., MUT in focal class and test case in test class) that allows to leverage the abilities from pre-training, as opposed to simply concatenating pieces of code. We therefore hypothesize that placing compressed context at the correct position could also benefit CAPT significantly.

[1] https://pitest.org/

# 7 Related Work

In this chapter, we describe the most relevant prior works and elaborate on how our work relates to them. We present related work on neural test case generation in section 7.1 and related work on soft prompting in code tasks in section 7.2. Finally, we describe prior works related to CAPT in section 7.3.

## 7.1 Neural Test Case Generation

In recent years, several works have applied neural models, in particular LMs, to the task of test case generation. In the following we describe three major directions of research: fine-tuning relatively small models, few-shot prompting large models and combining traditional with neural approaches.

Tufano et al. introduce AthenaTest[47], a state-of-the-art model based on the BART architecture [30] with 400M parameters and a maximum sequence length of 1024. First, the authors pre-train on a corpus of English text, further pre-train on a corpus of Java code and finally fine-tune on the test case generation dataset Methods2Test [46]. They find that both natural language and code pre-training benefits the final performance significantly. The overall training setup is very similar to ours, as we also use a 400M parameter model that is pre-trained on code and fine-tuned on test case generation. In contrast, our model is a decoder-only transformer with a maximum sequence length of 2048. The authors experiment with providing the name, constructor signatures, public method signatures and public fields of the focal class as additional context. The results show that while the biggest improvements stem from the class name and constructor signatures, using as much context as possible performs best in terms of validation loss. While we also observe improvements from additional context, we find most improvements stem from test class context and that focal class context gives only marginal further improvements. To further evaluate the model, the authors generate test cases for five projects from Defects4J [27]. They find that AthenaTest is able to generate correct tests for 43.75% of the MUTs while 16.21% of generated test cases are correct. In comparison, our model achieves a much lower number of correct test cases (e.g. 1.42% for `PT-Hard-All`). We argue that this is to a large degree due to the different evaluation setups. Whereas our model has to generate test cases that fit into an existing test class, the test classes are constructed around the test cases generated by AthenaTest. Unfortunately, the authors do not provide any details on this process and to what extent it is manual. The coverage achieved by AthenaTest is evaluated on a single focal class with 18 MUTs and compared to GPT-3 and Evosuite. The results show that AthenaTest performs best overall, closely followed by Evosuite. However, all methods achieve a rather low line coverage of well below 10% for almost all MUTs.

In contrast, our best neural model achieves an average line coverage of 12.21% whereas Evosuite achieves 37.78%. This large discrepancy in results confirms the importance of large-scale coverage evaluation. Motivated by the fact that the weights and code for AthenaTest are not publicly available, Alagarsamy et al. [2] attempted to replicate the study. With the training setup provided in the original paper, the authors are not able to replicate the results and achieve 0% correct test cases. However, through several modifications, they manage to generate 18.08% correct test cases on the same subset of Defects4J and call their model A3Test. The modifications include less natural language pre-training, additional pre-training on assertion generation and improved post-processing. Unlike AthenaTest, A3Test is not trained from scratch but based on the pre-trained PLBART model [1]. Since the pre-training corpus of this model is not publicly available, it cannot be excluded that data leakage from the Defects4J subset into the train set has occurred. The authors publish the code to replicate the results, but like Tufano et al. also don't publish the weights of the trained models. AthenaTest and A3Test are the most similar works to ours in the literature, whereas we differentiate from their work in three main ways. First, in Tests4J, we focus on fully automated large-scale coverage evaluation while specifically avoiding data leakage. We are also the first to provide details on the execution-based evaluation, in particular how test classes are constructed for generated test cases. Second, we directly compare neural models to Evosuite on Tests4J. Whereas Tufano et al. find that AthenaTest outperforms Evosuite in an evaluation on a single focal class, our large-scale evaluation finds that neural models are not yet comparable in terms of coverage. Lastly, additionally to providing context information in the hard prompt, we also experiment with CAPT in order to compress context and thus reduce memory and compute requirements.

Bareiß et al. [6] take a vastly different approach. Among other tasks, they evaluate the much larger Codex [12] model with 12B parameters in a few-shot setting on test case generation. That is, the model is not fine-tuned on the target task but merely provided a task description and a number of examples (pairs of MUT and test case) in the prompt. They compare their approach to Evosuite [19] and Randoop [36] on 18 MUTs from 9 projects with respect to coverage. They find that Codex outperforms Randoop (14% vs 12%) but under performs Evosuite (14% vs 16%). They also find that combining the three approaches significantly increases the coverage (25%), indicating that the approaches are complementary and test different parts of the MUTs. Schäfer et al. [43] also leverage Codex [12] to generate test cases for JavaScript packages. In contrast to the previously discussed approaches, the authors use usage examples from the documentation as few-shot examples. Furthermore, they implement an adaptive component that executes generated tests and re-prompts the model with the failing test case and the error message. In an evaluation on 1684 MUTs, the approach achieves a median statement coverage of 68.2%. In this work, we don't include Codex or other large model APIs in the evaluation mainly because the proposed method CAPT requires backpropagating gradients through the model, which is not possible with current APIs. Future work should evaluate a broader spectrum of approaches on Tests4J, including the two aforementioned Codex-based methods.

Lemieux et al. [28] combine LM-based approaches with search-based approaches similar to Evosuite and present CodaMosa, a hybrid test case generator for Python. In particular, they

augment the evolutionary algorithm MOSA [39] with Codex [12] in order to escape local optima. That is, when the evolutionary search fails to further improve coverage results, they prompt Codex to generate test cases and use them as additional seeds. In an evaluation on 35 Python projects, they find that the approach significantly outperforms both MOSA and Codex on their own, as well as the combined test suites of both methods.

## 7.2 Soft Prompt Tuning in Code Tasks

Whereas soft prompt tuning methods [31, 29, 33] were originally proposed in the domain of natural language, they have also been applied to several code-related tasks.

Zlotchevski et al. [54] leverage prefix tuning [31] for personalized code generation. The general idea is to adjust the model to every specific project. The resources required for storing and deploying a separate model copy per project motivates the need for parameter-efficient tuning methods. The authors train a set of prefix tokens for every project on the test case generation task and evaluate the performance against tuning selected layers of the model and to full fine-tuning. The evaluation does not include execution-based metrics, but several intrinsic metrics, such as exact match, BLEU and code style match. The results show that prefix tuning performs similarly to both tuning only the last decoder layer in terms of exact match, BLEU and perplexity, with a small gap towards fine-tuning. However, the authors note that prefix tuning takes much longer to train and argue that this might be due to the random initialization of newly introduced parameters. The authors conclude that all personalization approaches lead to significant improvements over the baseline, and that all approaches have different merits. We argue that given our results in RQ5 and the small differences in performance between approaches, an execution-based evaluation might lead to further insights.

Wang et al. [49] attempt to investigate the effectiveness of prompt tuning [29] on code tasks. To that end, they evaluate it on defect prediction, code summarization and code translation with CodeBERT [18] and CodeT5 [50] on multiple programming languages. They find that prefix tuning consistently outperforms fine-tuning in almost all scenarios. They also note that the improvements become more apparent with smaller models and when data is scarce. Overall, the results are somewhat opposed to most other works in both NLP and code intelligence (including ours), where full fine-tuning usually is an upper bound, with prompt tuning coming very close at best.

He et al. [25] explore prefix tuning as a method for controllable generation in the code domain. In particular, they train two prefixes, one for generating secure code and one for generating vulnerable code. Additional to regular the language modelling loss, the authors propose two other losses for training the prefixes. A contrastive loss between the logits of the secure model and the logits of the vulnerable model is introduced to provide an explicit signal differentiating secure and vulnerable code. To minimize catastrophic forgetting caused by the contrastive loss, the authors further introduce a KL-divergence loss between the logits of the secure model and the base language model. Using this setup, prefix tuning clearly outperforms full fine-tuning. However, the setups are not directly comparable because the special losses are only applied to the prefix tuning model. The authors further find that 5 prefix tokens work best (we use 10) and that scaling the model

size does not necessarily yield better results. In particular, the results show that among the 350M, 2.7B and 6.1B models, the 2.7B model performs best, whereas 350M and 6.1B perform similarly. In this work, we did not perform formal experiments with different model sizes, but preliminary experiments have shown similar results.

## 7.3 Dynamic Prompt Tuning and Multi-modal Language Models

In this work, we propose CAPT, a novel method for processing long contexts with relaxed scaling properties. It is essentially derivative of two main lines of research: dynamic prompt tuning and multi-modal language models.

Related work on dynamic prompt tuning has found that making soft prompts depend on the input can slightly improve performance in several NLP tasks. In that sense, the setup is slightly different from ours. Whereas we encode additional context information and use it as soft prompt, dynamic prompt tuning encodes the current input itself uses the resulting embedding as soft prompt. Consequently, the motivation also differs from ours: we aim to reduce computational requirements through compression, whereas dynamic prompt tuning seeks more optimal soft prompts. However, the related work is still very relevant because it served as initial inspiration and can provide a few insights on how to design the architecture. Tang et al. [45] propose context tuning and evaluate it on story generation, opinion generation and text summarization. In context tuning, a number of mask tokens are both prepended and appended to the input and fed through BERT. For each of the mask tokens, the language modelling head is used to obtain the probability distribution for this token. The distribution is then used to create the final soft prompt as a weighted sum over all the token embeddings of BERT. These soft prompts are inserted at the input layer only. In the downstream evaluation, they show competitive performance of the approach, outperforming vanilla prompt tuning, prefix tuning and even full fine-tuning in most setups. The approach is mainly interesting to CAPT because of the prompt generator architecture. Exploring this setup could be a promising direction for future research. In particular, the approach of using mask tokens as embedding sources could in theory alleviate the scaling problem discussed in section 6.2. Wu et al. [51] propose Instance-Dependent Prompt Generation (IDPG), another dynamic prompt tuning method. Here, the prompt generator is the LM itself. More specifically, the LM is used to obtain a holistic representation of the input. Next, the embedding is fed through a bottleneck layer that first projects it down and then up to the target dimensionality, which is number of layers times number of soft tokens times hidden dimensionality. The approach is evaluated on multiple NLU tasks and found to be comparable with full fine-tuning. This approach is the most similar to ours in the dynamic prompt tuning literature in the sense that the soft tokens are injected at every layer and that all information flows through a single holistic representation that is afterward projected up to the target size. Whereas the results of IDPG indicate that this setup works quite well for dynamic prompt tuning, we find that restricting the information flow to a single holistic embedding is problematic when injecting additional context information.

Related work on multi-modal language models is also closely related to CAPT. Hao et al. [24] introduce MetaLM, a general-purpose architecture and philosophy for multi-modal modelling. The

authors propose to relax the definition of a token and let tokens represent arbitrary information. This allows architectures that were originally designed to process text to process any modality that can be encoded by a neural network. In the experiments, the authors focus on text and images as modalities. Additional information, like an image, is first tokenized and processed by an encoder. The resulting tokens representations are then projected into the target latent space by a linear connector module. The tokens are only injected at the input layer. In a follow-up paper, researchers from the same group introduce Kosmos-1 [26], a multi-modal LM based on the MetaLM architecture, trained from scratch on large-scale corpora of text and images. It is instruction-tuned, capable of in-context learning, and outperforms much larger baselines. The mechanism is similar to ours in the sense that both inject additional context information into the model via soft tokens. The main difference is that our motivation is compression and associated memory savings, whereas MetaLM and Kosmos-1 enable multi-modal inputs with language models. The input is not compressed, i.e., if additional text is encoded and injected (e.g. text in a language not supported by the main LM), the soft prompt has the same length as the original text.

In summary, we see CAPT as a combination of methods from these two lines of research, albeit motivated by compression rather than mere task performance or multi-modality. Our approach shares with context tuning and IDPG that the input of the prompt generator in transformed into a fixed number of soft tokens, whereas the number of soft tokens is not altered by encoders in MetaLM and Kosmos-1. On the other hand, our approach shares with MetaLM and Kosmos-1 that the input to the prompt generator is additional context beyond the hard prompt to the LM. Both lines of research serve as inspiration and provide interesting insights. Yet, it remains an open question whether using injection at the input layer or at every layer is more beneficial.

# 8 Conclusion

This work revolves around the task of test case generation. We first identified the gap of missing benchmarks for test case generators that are designed with LMs in mind, avoid data leakage from popular pre-training datasets and provide both intrinsic and execution-based evaluations. To fill that gap, we proposed Tests4J. It includes 12k pairs of MUTs and test cases from 60 Java projects. Out of these 60 projects, 19 were randomly selected for evaluation, while the rest is used as training data. The 19 evaluation projects were set up to be executable in a fully automatic way. Therefore, Tests4J allows researchers to save generated tests into a JSON file with a simple schema and run coverage analyses in a single command. This will output coverage information per MUT, per repository and overall, as well as compilation feedback per test case and BLEU scores. We present directions for future work, including dynamically adjusted or constructed test classes, further increases in scale and the integration of mutation scores.

We integrated Evosuite as a strong non-neural baseline. In a comparison with our best neural model, we found that individual test cases of the two approaches achieved similar coverage. However, Evosuite was able to generate three times as many executable test cases and covered three times as many lines in total. In the light of this insight, we believe that neural models could become competitive in the future, for instance using dynamically adjusted test classes and by sampling more test cases from the trained LM. Furthermore, we see compatibility as a major advantage of LMs, given that Evosuite failed to generate test cases on 4 out of the 11 projects in the test set.

Based on Tests4J, we experimented with different training setups for neural test case generators based on PolyCoder, a pre-trained LM of code with 400M parameters. First, we compared prefix tuning to zero-shot prompting and full fine-tuning. We found that prefix tuning achieves significant improvements over zero-shot, but lacks behind full fine-tuning in terms of coverage, number of correct test cases and other criteria. Next, we investigated the effects of context information on the performance. We found that context is essential, improving line coverage by more than 4x. We furthermore found that test class context was more effective than focal class context, and that the two are highly complementary. We conclude that adding even more context will likely yield additional improvements, further motivating the usage of models that can process sequences beyond the limit of 2048 tokens and without quadratic complexity. We proposed and evaluated CAPT as such a method, but found that it did not achieve significantly better results than not using the context at all. In the discussion, we present hypotheses for the reasons but leave further experimentation to future work. Lastly, we analyze the results across models and projects. We find that loss is not an ideal indicator and that there is a high variance in coverage between projects, and thus advocate for large-scale execution-based evaluations.

# A Appendix

## A.1 Maven Configuration

### A.1.1 Maven Compiler Plugin Configuration

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.10.1</version>
    <configuration>
        <failOnError>false</failOnError>
            <compilerArgs>
                <arg>-Xmaxerrs</arg>
                <arg>9999999</arg>
            </compilerArgs>
    </configuration>
</plugin>
```

### A.1.2 Maven Surefire Plugin Configuration

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M7</version>
    <configuration>
        <testFailureIgnore>true</testFailureIgnore>
    </configuration>
</plugin>
```

### A.1.3 Maven JaCoCo Plugin Configuration

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.8</version>
    <executions>
        <execution>
```

```
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>test</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

### A.1.4 Maven Compile Log Regex

```
COMPILE_REGEX_1 = re.compile(r"\[ERROR\] (.*):\[(\d+)(,\d+)?\] (.*)")
COMPILE_REGEX_2 = re.compile(r"(/.*):\[(\d+)(,\d+)?\] error: (.*)")
COMPILE_REGEX_3 = re.compile(r"\[WARNING\] (.*):\[(\d+)(,\d+)?\] error: (.*)")
```

### A.1.5 Maven Execution Log Regex

```
MAVEN_TEST_ERROR_REGEX = re.compile(
    r"\[ERROR\] ([a-zA-Z_$][a-zA-Z_$0-9\.]*)  Time elapsed.*\n(.*)\n"
)
MAVEN_TEST_ERROR_REGEX_2 = re.compile(
    r"\[ERROR\] .* InvalidTestClass Invalid test class '(.*)':\n  1\. Method
    ↪  ([a-zA-Z_$][a-zA-Z_$0-9\.]*)(\(\))? (.*)\n",
    re.MULTILINE,
)
MAVEN_TEST_ERROR_REGEX_3 = re.compile(
    r"\[ERROR\] ([a-zA-Z_$][a-zA-Z_$0-9\.]*)  Time elapsed.* <<< (FAILURE|ERROR)!\n(.*)\n",
    re.MULTILINE,
)
```

## A.2 List of Repositories

| Split | Repository ID | Commit Hash | Java Version | #Test Cases |
|---|---|---|---|---|
| Train | phax/ph-commons | b121f7f | 11 | 1000 |
| Train | openaire/iis | bec54e8 | 8 | 708 |
| Train | jdemetra/jdemetra-core | 9f41c8c | 8 | 489 |
| Train | toolisticon/annotation-processor-toolkit | fdbce50 | 8 | 435 |
| Train | coffee4j/coffee4j | edb9d7b | 11 | 343 |

| | | | | |
|---|---|---|---|---|
| Train | Johnnei/JavaTorrent | 690b2ed | 11 | 337 |
| Train | guardtime/ksi-java-sdk | de75695 | 8 | 327 |
| Train | seeker/similarImage | 32dd2c8 | 11 | 319 |
| Train | charite/jannovar | 87f3a92 | 8 | 315 |
| Train | knowsys/rulewerk | 252837c | 8 | 274 |
| Train | eeverman/andhow | d632675 | 8 | 272 |
| Train | sane-city/wot-servient | a3cec2c | 11 | 251 |
| Train | AmadeusITGroup/HttpSessionReplacer | 16e9d93 | 8 | 248 |
| Train | adorsys/xs2a-connector-examples | 47f1d0a | 11 | 223 |
| Train | dan2097/opsin | 3cf9a3a | 8 | 220 |
| Train | yandex/yoctodb | a4e91c5 | 8 | 197 |
| Train | savoirtech/hecate | eafcdde | 8 | 170 |
| Train | jencisopy/JavaBeanStack | 2923542 | 8 | 154 |
| Train | johnmay/beam | d79481f | 8 | 153 |
| Train | minijax/minijax | 5fdb902 | 11 | 146 |
| Train | apache/sling-org-apache-sling-app-cms | 7363eb6 | 11 | 132 |
| Train | scientific-tool-set/scitos | decca29 | 11 | 129 |
| Train | verhas/javageci | d91952b | 11 | 125 |
| Train | btc-ag/redg | a2d7284 | 8 | 118 |
| Train | NaluKit/nalu | 648648b | 8 | 113 |
| Train | angelozerr/lsp4xml | b11ed6b | 8 | 111 |
| Train | opencb/biodata | 8c9e4c3 | 8 | 110 |
| Train | gbif/checklistbank | c2c8bb9 | 8 | 103 |
| Train | salesforce/pyplyn | e70aab5 | 8 | 103 |
| Train | heremaps/here-aaa-java-sdk | 20de9d6 | 8 | 102 |
| Train | pentaho/pentaho-commons-xul | 1a86f72 | 11 | 100 |
| Train | difi/vefa-peppol | e3c2b07 | 8 | 98 |
| Train | JavatarPro/javatar-security | 0ffafb8 | 8 | 97 |
| Train | AxonFramework/extension-kafka | a2c69c9 | 8 | 97 |
| Train | wso2/carbon-registry | 7820880 | 8 | 96 |
| Train | BlackPepperSoftware/bowman | 6e49b6e | 8 | 94 |
| Train | ehrmann/vcdiff-java | c968244 | 8 | 93 |
| Train | asad/SMSD | 173a3fc | 8 | 87 |
| Train | difi/oxalis | 8943091 | 8 | 83 |
| Train | tomtom-international/openlr | 9e7bae3 | 8 | 81 |
| Train | apache/empire-db | 86624c4 | 8 | 80 |
| Val | intuit/CloudRaider | e12258b | 8 | 231 |
| Val | ARMmbed/java-coap | ad9f715 | 8 | 223 |
| Val | eclipse/winery | 1c0f66c | 8 | 167 |
| Val | itsallcode/openfasttrace | 83a8063 | 11 | 149 |
| Val | HDouss/jeometry | 7fc344e | 8 | 140 |

| | | | | |
|---|---|---|---|---|
| Val | kermoss/kermoss | 8e7a3bb | 8 | 116 |
| Val | InteractiveAdvertisingBureau/iabtcf-java | 14f7e37 | 8 | 96 |
| Val | leancloud/filter-service | e9e44b8 | 8 | 82 |
| Val | Domo42/saga-lib | aa0f234 | 8 | 81 |
| Test | gooddata/gooddata-java | 840e8ff | 11 | 377 |
| Test | shroffk/phoebus | 8435525 | 11 | 285 |
| Test | SonarSource/orchestrator | d6824b3 | 11 | 249 |
| Test | messaginghub/pooled-jms | 6a1b942 | 11 | 240 |
| Test | intuit/QuickBooks-V3-Java-SDK | 0486ee9 | 8 | 239 |
| Test | adorsys/XS2A-Sandbox | ff6a12d | 11 | 180 |
| Test | Flipkart/foxtrot | 2f062e4 | 8 | 175 |
| Test | bazaarvoice/ostrich | 09e85e0 | 8 | 163 |
| Test | nhl/dflib | 6f960ba | 11 | 134 |
| Test | monarch-initiative/phenol | 0da2c42 | 11 | 120 |
| Test | criteo/garmadon | cfe0aa1 | 8 | 109 |
| Test | flipkart-incubator/Lyrics | c65df2e | 8 | 103 |
| Test | seedstack/seed | de8c0be | 11 | 97 |

Table A.1: List of all Repositories in the Dataset. The Repository ID corresponds to organisation/name.

## A.3 CAPT Result Tables

### A.3.1 BERT

PT-Hard-None-BERT-TCl loss: 0.8944 BLEU: 0.0557 crystalBLEU: 0.0257

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1185 | 956 (80.68%) | 83 (7.00%) | 82 (6.92%) | 1 (0.08%) |
| bazaarvoice-ostrich | 0.88% | 1.41% | 0.79% | 1.41% | 1394 | 1199 (86.01%) | 29 (2.08%) | 27 (1.94%) | 5 (0.36%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2980 | 2670 (89.60%) | 409 (13.72%) | 357 (11.98%) | 6 (0.20%) |
| flipkart-incubator-Lyrics | 3.09% | 1.21% | 4.76% | 0.83% | 1500 | 1027 (68.47%) | 46 (3.07%) | 29 (1.93%) | 2 (0.13%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 986 | 877 (88.95%) | 32 (3.25%) | 31 (3.14%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 1.53% | 1.58% | 0.74% | 1.64% | 2242 | 1970 (87.87%) | 92 (4.10%) | 85 (3.79%) | 1 (0.04%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2847 | 2723 (95.64%) | 114 (4.00%) | 112 (3.93%) | 1 (0.04%) |
| shroffk-phoebus | 0.47% | 1.32% | 0.25% | 1.32% | 3405 | 2598 (76.30%) | 132 (3.88%) | 97 (2.85%) | 13 (0.38%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1748 | 1593 (91.13%) | 30 (1.72%) | 28 (1.60%) | 3 (0.17%) |
| nhl-dflib | 3.50% | 3.64% | 3.59% | 3.44% | 1652 | 1357 (82.14%) | 53 (3.21%) | 45 (2.72%) | 2 (0.12%) |
| monarch-initiative-phenol | 10.87% | 6.87% | 10.06% | 7.17% | 1549 | 1208 (77.99%) | 77 (4.97%) | 56 (3.62%) | 7 (0.45%) |
| Total | 1.79% | 1.73% | 1.63% | 1.65% | 21488 | 18178 (84.60%) | 1097 (5.11%) | 949 (4.42%) | 41 (0.19%) |

Table A.2: Results for PT-Hard-None-BERT-TCl, where the test class is encoded with BERT into 1 soft token.

PT-Hard-None-BERT-TCl loss: 0.8937 BLEU: 0.0586 crystalBLEU: 0.0285

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1123 | 910 (81.03%) | 55 (4.90%) | 54 (4.81%) | 0 (0.00%) |
| bazaarvoice-ostrich | 0.88% | 1.41% | 0.79% | 1.41% | 1278 | 1103 (86.31%) | 16 (1.25%) | 11 (0.86%) | 3 (0.23%) |
| adorsys-XS2A-Sandbox | 0.30% | 0.51% | 0.48% | 0.38% | 2901 | 2568 (88.52%) | 329 (11.34%) | 268 (9.24%) | 4 (0.14%) |
| flipkart-incubator-Lyrics | 1.93% | 0.76% | 1.59% | 0.28% | 1364 | 1028 (75.37%) | 46 (3.37%) | 28 (2.05%) | 1 (0.07%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 851 | 742 (87.19%) | 20 (2.35%) | 20 (2.35%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2239 | 1940 (86.65%) | 60 (2.68%) | 60 (2.68%) | 1 (0.04%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2795 | 2612 (93.45%) | 62 (2.22%) | 59 (2.11%) | 0 (0.00%) |
| shroffk-phoebus | 0.79% | 1.97% | 0.89% | 1.73% | 3274 | 2457 (75.05%) | 131 (4.00%) | 73 (2.23%) | 15 (0.46%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1516 | 1419 (93.60%) | 23 (1.52%) | 19 (1.25%) | 1 (0.07%) |
| nhl-dflib | 3.23% | 2.31% | 3.19% | 2.11% | 1583 | 1306 (82.50%) | 53 (3.35%) | 36 (2.27%) | 4 (0.25%) |
| monarch-initiative-phenol | 15.22% | 10.64% | 12.43% | 11.52% | 1405 | 1103 (78.51%) | 64 (4.56%) | 45 (3.20%) | 9 (0.64%) |
| Total | 1.82% | 1.93% | 1.72% | 1.82% | 20329 | 17188 (84.55%) | 859 (4.23%) | 673 (3.31%) | 38 (0.19%) |

Table A.3: Results for PT-Hard-None-BERT-TCl, where the test class is encoded with BERT into 10 soft tokens.

PT-Hard-None-BERT_C-TCl loss: 0.9004 BLEU: 0.0611 crystalBLEU: 0.0304

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 0.00% | 0.00% | 0.00% | 0.00% | 1102 | 943 (85.57%) | 77 (6.99%) | 76 (6.90%) | 0 (0.00%) |
| bazaarvoice-ostrich | 4.87% | 7.04% | 3.97% | 7.04% | 1274 | 1129 (88.62%) | 25 (1.96%) | 23 (1.81%) | 3 (0.24%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2850 | 2645 (92.81%) | 279 (9.79%) | 249 (8.74%) | 6 (0.21%) |
| flipkart-incubator-Lyrics | 0.00% | 0.00% | 0.00% | 0.00% | 1284 | 1030 (80.22%) | 33 (2.57%) | 14 (1.09%) | 0 (0.00%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 821 | 679 (82.70%) | 17 (2.07%) | 17 (2.07%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2148 | 1958 (91.15%) | 96 (4.47%) | 92 (4.28%) | 4 (0.19%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2838 | 2653 (93.48%) | 94 (3.31%) | 91 (3.21%) | 6 (0.21%) |
| shroffk-phoebus | 0.79% | 1.97% | 0.38% | 1.97% | 3218 | 2469 (76.72%) | 170 (5.28%) | 98 (3.05%) | 20 (0.62%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1596 | 1481 (92.79%) | 27 (1.69%) | 20 (1.25%) | 0 (0.00%) |
| nhl-dflib | 9.97% | 2.12% | 7.57% | 1.68% | 1599 | 1380 (86.30%) | 45 (2.81%) | 34 (2.13%) | 0 (0.00%) |
| monarch-initiative-phenol | 15.22% | 10.64% | 12.43% | 11.52% | 1388 | 1090 (78.53%) | 52 (3.75%) | 34 (2.45%) | 8 (0.58%) |
| Total | 2.03% | 2.17% | 1.72% | 2.13% | 20118 | 17457 (86.77%) | 915 (4.55%) | 748 (3.72%) | 47 (0.23%) |

Table A.4: Results for PT-Hard-None-BERT_C-TCl, where the test class is encoded with BERT_C into 30 soft tokens.

PT-Hard-None-BERT_C-TCl loss: 0.9051 BLEU: 0.0654 crystalBLEU: 0.0328

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1178 | 962 (81.66%) | 138 (11.71%) | 136 (11.54%) | 2 (0.17%) |
| bazaarvoice-ostrich | 4.87% | 7.04% | 3.97% | 7.04% | 1352 | 1149 (84.99%) | 33 (2.44%) | 25 (1.85%) | 7 (0.52%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2926 | 2714 (92.75%) | 352 (12.03%) | 312 (10.66%) | 14 (0.48%) |
| flipkart-incubator-Lyrics | 0.00% | 0.00% | 0.00% | 0.00% | 1338 | 1029 (76.91%) | 36 (2.69%) | 23 (1.72%) | 1 (0.07%) |
| seedstack-seed | 1.09% | 2.13% | 0.52% | 2.13% | 901 | 738 (81.91%) | 26 (2.89%) | 25 (2.77%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.34% | 1.56% | 0.50% | 1.17% | 2240 | 1995 (89.06%) | 87 (3.88%) | 74 (3.30%) | 4 (0.18%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2941 | 2761 (93.88%) | 126 (4.28%) | 123 (4.18%) | 4 (0.14%) |
| shroffk-phoebus | 0.63% | 1.32% | 0.25% | 1.32% | 3184 | 2409 (75.66%) | 131 (4.11%) | 69 (2.17%) | 15 (0.47%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1662 | 1528 (91.94%) | 34 (2.05%) | 33 (1.99%) | 5 (0.30%) |
| nhl-dflib | 3.23% | 2.31% | 3.19% | 2.11% | 1526 | 1312 (85.98%) | 28 (1.83%) | 24 (1.57%) | 3 (0.20%) |
| monarch-initiative-phenol | 15.22% | 10.64% | 12.43% | 11.52% | 1332 | 1083 (81.31%) | 55 (4.13%) | 45 (3.38%) | 2 (0.15%) |
| Total | 1.92% | 2.54% | 1.66% | 2.51% | 20580 | 17680 (85.91%) | 1046 (5.08%) | 889 (4.32%) | 57 (0.28%) |

Table A.5: Results for PT-Hard-None-BERT_C-TCl, where the test class is encoded with BERT_C into 90 soft tokens.

PT-Hard-None-BERT_C-TCl loss: 1.1792 BLEU: 0.0627 crystalBLEU: 0.0306

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1173 | 972 (82.86%) | 117 (9.97%) | 112 (9.55%) | 0 (0.00%) |
| bazaarvoice-ostrich | 0.88% | 1.41% | 0.79% | 1.41% | 1362 | 1170 (85.90%) | 31 (2.28%) | 23 (1.69%) | 5 (0.37%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2831 | 2657 (93.85%) | 305 (10.77%) | 269 (9.50%) | 7 (0.25%) |
| flipkart-incubator-Lyrics | 0.00% | 0.00% | 0.00% | 0.00% | 1347 | 1043 (77.43%) | 58 (4.31%) | 34 (2.52%) | 0 (0.00%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 866 | 743 (85.80%) | 16 (1.85%) | 16 (1.85%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 1.53% | 1.58% | 0.74% | 1.64% | 2286 | 2086 (91.25%) | 101 (4.42%) | 90 (3.94%) | 2 (0.09%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2985 | 2764 (92.60%) | 105 (3.52%) | 93 (3.12%) | 1 (0.03%) |
| shroffk-phoebus | 0.47% | 1.32% | 0.25% | 1.32% | 3237 | 2498 (77.17%) | 169 (5.22%) | 103 (3.18%) | 20 (0.62%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1649 | 1538 (93.27%) | 37 (2.24%) | 30 (1.82%) | 6 (0.36%) |
| nhl-dflib | 2.96% | 0.98% | 2.79% | 0.78% | 1616 | 1357 (83.97%) | 33 (2.04%) | 19 (1.18%) | 0 (0.00%) |
| monarch-initiative-phenol | 15.22% | 10.64% | 12.43% | 11.52% | 1433 | 1152 (80.39%) | 63 (4.40%) | 42 (2.93%) | 6 (0.42%) |
| Total | 1.89% | 1.72% | 1.50% | 1.73% | 20785 | 17980 (86.50%) | 1035 (4.98%) | 831 (4.00%) | 47 (0.23%) |

Table A.6: Results for PT-Hard-None-BERT_C-TCl, where the test class is encoded with BERT_C into 150 soft tokens.

PT-Hard-None-BERT_C-TCl loss: 1.1810 BLEU: 0.0632 crystalBLEU: 0.0312

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 0.00% | 0.00% | 0.00% | 0.00% | 1181 | 1020 (86.37%) | 136 (11.52%) | 133 (11.26%) | 6 (0.51%) |
| bazaarvoice-ostrich | 4.87% | 7.04% | 3.97% | 7.04% | 1418 | 1213 (85.54%) | 28 (1.97%) | 21 (1.48%) | 7 (0.49%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2817 | 2687 (95.39%) | 324 (11.50%) | 289 (10.26%) | 16 (0.57%) |
| flipkart-incubator-Lyrics | 2.32% | 0.91% | 2.38% | 0.42% | 1425 | 1052 (73.82%) | 44 (3.09%) | 24 (1.68%) | 2 (0.14%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 910 | 725 (79.67%) | 31 (3.41%) | 30 (3.30%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2279 | 2094 (91.88%) | 88 (3.86%) | 79 (3.47%) | 4 (0.18%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2968 | 2793 (94.10%) | 176 (5.93%) | 173 (5.83%) | 6 (0.20%) |
| shroffk-phoebus | 1.02% | 2.47% | 0.89% | 2.30% | 3235 | 2575 (79.60%) | 160 (4.95%) | 98 (3.03%) | 29 (0.90%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1665 | 1558 (93.57%) | 32 (1.92%) | 24 (1.44%) | 4 (0.24%) |
| nhl-dflib | 3.50% | 3.64% | 3.59% | 3.44% | 1558 | 1348 (86.52%) | 47 (3.02%) | 30 (1.93%) | 2 (0.13%) |
| monarch-initiative-phenol | 15.22% | 10.64% | 12.43% | 11.52% | 1406 | 1128 (80.23%) | 54 (3.84%) | 42 (2.99%) | 3 (0.21%) |
| Total | 1.79% | 2.44% | 1.63% | 2.35% | 20862 | 18193 (87.21%) | 1120 (5.37%) | 943 (4.52%) | 79 (0.38%) |

Table A.7: Results for PT-Hard-None-BERT_C-TCl, where the test class is encoded with BERT_C into 188 soft tokens.

### A.3.2 LSTM

PT-Hard-None-LSTM-TCl loss: 0.8957 BLEU: 0.0587 crystalBLEU: 0.0281

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1107 | 877 (79.22%) | 52 (4.70%) | 50 (4.52%) | 0 (0.00%) |
| bazaarvoice-ostrich | 4.87% | 7.04% | 3.97% | 7.04% | 1292 | 1085 (83.98%) | 28 (2.17%) | 17 (1.32%) | 3 (0.23%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 3018 | 2747 (91.02%) | 400 (13.25%) | 342 (11.33%) | 7 (0.23%) |
| flipkart-incubator-Lyrics | 0.00% | 0.00% | 0.00% | 0.00% | 1370 | 989 (72.19%) | 44 (3.21%) | 22 (1.61%) | 0 (0.00%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 902 | 784 (86.92%) | 33 (3.66%) | 32 (3.55%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2231 | 1927 (86.37%) | 68 (3.05%) | 67 (3.00%) | 0 (0.00%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2622 | 2513 (95.84%) | 78 (2.97%) | 77 (2.94%) | 1 (0.04%) |
| shroffk-phoebus | 0.39% | 1.15% | 0.63% | 0.99% | 3349 | 2610 (77.93%) | 138 (4.12%) | 76 (2.27%) | 17 (0.51%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1577 | 1474 (93.47%) | 25 (1.59%) | 18 (1.14%) | 3 (0.19%) |
| nhl-dflib | 3.50% | 3.64% | 3.59% | 3.44% | 1525 | 1289 (84.52%) | 45 (2.95%) | 31 (2.03%) | 4 (0.26%) |
| monarch-initiative-phenol | 10.05% | 6.00% | 9.47% | 5.72% | 1411 | 1137 (80.58%) | 66 (4.68%) | 33 (2.34%) | 2 (0.14%) |
| Total | 1.49% | 1.95% | 1.56% | 1.80% | 20404 | 17432 (85.43%) | 977 (4.79%) | 765 (3.75%) | 37 (0.18%) |

Table A.8: Results for PT-Hard-None-LSTM_C-TCl, where the test class is encoded with LSTM into 1 soft token.

PT-Hard-None-LSTM-TCl loss: 0.8908 BLEU: 0.0618 crystalBLEU: 0.0305

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1159 | 911 (78.60%) | 77 (6.64%) | 75 (6.47%) | 0 (0.00%) |
| bazaarvoice-ostrich | 4.42% | 7.04% | 3.97% | 7.04% | 1213 | 1054 (86.89%) | 29 (2.39%) | 17 (1.40%) | 2 (0.16%) |
| adorsys-XS2A-Sandbox | 0.30% | 0.51% | 0.48% | 0.38% | 2889 | 2620 (90.69%) | 338 (11.70%) | 284 (9.83%) | 5 (0.17%) |
| flipkart-incubator-Lyrics | 0.00% | 0.00% | 0.00% | 0.00% | 1329 | 952 (71.63%) | 36 (2.71%) | 24 (1.81%) | 3 (0.23%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 863 | 723 (83.78%) | 25 (2.90%) | 25 (2.90%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2227 | 1923 (86.35%) | 66 (2.96%) | 61 (2.74%) | 7 (0.31%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2763 | 2617 (94.72%) | 80 (2.90%) | 73 (2.64%) | 3 (0.11%) |
| shroffk-phoebus | 0.47% | 1.32% | 0.25% | 1.32% | 3241 | 2445 (75.44%) | 129 (3.98%) | 68 (2.10%) | 13 (0.40%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1536 | 1430 (93.10%) | 31 (2.02%) | 26 (1.69%) | 2 (0.13%) |
| nhl-dflib | 3.23% | 2.31% | 3.19% | 2.11% | 1586 | 1314 (82.85%) | 31 (1.95%) | 18 (1.13%) | 1 (0.06%) |
| monarch-initiative-phenol | 15.22% | 10.64% | 12.43% | 11.52% | 1381 | 1097 (79.44%) | 64 (4.63%) | 50 (3.62%) | 9 (0.65%) |
| Total | 1.81% | 2.31% | 1.63% | 2.27% | 20187 | 17086 (84.64%) | 906 (4.49%) | 721 (3.57%) | 45 (0.22%) |

Table A.9: Results for PT-Hard-None-LSTM-TCl, where the test class is encoded with LSTM into 10 soft tokens.

PT-Hard-None-LSTM_C-TCl loss: 0.8964 BLEU: 0.0644 crystalBLEU: 0.0322

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 0.00% | 0.00% | 0.00% | 0.00% | 1171 | 997 (85.14%) | 83 (7.09%) | 82 (7.00%) | 1 (0.09%) |
| bazaarvoice-ostrich | 10.62% | 11.27% | 7.14% | 9.15% | 1375 | 1192 (86.69%) | 37 (2.69%) | 27 (1.96%) | 6 (0.44%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2950 | 2796 (94.78%) | 325 (11.02%) | 283 (9.59%) | 12 (0.41%) |
| flipkart-incubator-Lyrics | 0.00% | 0.00% | 0.00% | 0.00% | 1320 | 1012 (76.67%) | 51 (3.86%) | 26 (1.97%) | 1 (0.08%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 878 | 713 (81.21%) | 18 (2.05%) | 18 (2.05%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2268 | 2033 (89.64%) | 77 (3.40%) | 71 (3.13%) | 10 (0.44%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 3143 | 2969 (94.46%) | 137 (4.36%) | 131 (4.17%) | 8 (0.25%) |
| shroffk-phoebus | 0.47% | 1.32% | 0.25% | 1.32% | 3218 | 2499 (77.66%) | 164 (5.10%) | 96 (2.98%) | 12 (0.37%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1760 | 1700 (96.59%) | 51 (2.90%) | 37 (2.10%) | 0 (0.00%) |
| nhl-dflib | 0.00% | 0.00% | 0.00% | 0.00% | 1555 | 1306 (83.99%) | 25 (1.61%) | 19 (1.22%) | 0 (0.00%) |
| monarch-initiative-phenol | 14.67% | 10.16% | 11.83% | 11.23% | 1412 | 1161 (82.22%) | 53 (3.75%) | 42 (2.97%) | 0 (0.00%) |
| Total | 1.55% | 2.26% | 1.17% | 2.08% | 21050 | 18378 (87.31%) | 1021 (4.85%) | 832 (3.95%) | 50 (0.24%) |

Table A.10: Results for PT-Hard-None-LSTM_C-TCl, where the test class is encoded with LSTM_C into 30 soft tokens.

PT-Hard-None-LSTM_C-TCl loss: 0.8941 BLEU: 0.0608 crystalBLEU: 0.0295

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1117 | 894 (80.04%) | 52 (4.66%) | 52 (4.66%) | 0 (0.00%) |
| bazaarvoice-ostrich | 10.62% | 11.27% | 7.14% | 9.15% | 1261 | 1064 (84.38%) | 31 (2.46%) | 25 (1.98%) | 8 (0.63%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2731 | 2522 (92.35%) | 301 (11.02%) | 239 (8.75%) | 7 (0.26%) |
| flipkart-incubator-Lyrics | 0.00% | 0.00% | 0.00% | 0.00% | 1427 | 1013 (70.99%) | 55 (3.85%) | 39 (2.73%) | 1 (0.07%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 864 | 734 (84.95%) | 24 (2.78%) | 23 (2.66%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2251 | 1955 (86.85%) | 75 (3.33%) | 75 (3.33%) | 1 (0.04%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2744 | 2617 (95.37%) | 113 (4.12%) | 110 (4.01%) | 2 (0.07%) |
| shroffk-phoebus | 0.71% | 1.81% | 0.76% | 1.64% | 3220 | 2452 (76.15%) | 154 (4.78%) | 85 (2.64%) | 22 (0.68%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1496 | 1438 (96.12%) | 33 (2.21%) | 24 (1.60%) | 3 (0.20%) |
| nhl-dflib | 0.54% | 2.67% | 0.80% | 2.67% | 1543 | 1290 (83.60%) | 44 (2.85%) | 35 (2.27%) | 3 (0.19%) |
| monarch-initiative-phenol | 17.66% | 14.99% | 14.20% | 15.87% | 1276 | 1048 (82.13%) | 60 (4.70%) | 46 (3.61%) | 7 (0.55%) |
| Total | 2.03% | 3.12% | 1.76% | 2.90% | 19930 | 17027 (85.43%) | 942 (4.73%) | 753 (3.78%) | 54 (0.27%) |

Table A.11: Results for PT-Hard-None-LSTM_C-TCl, where the test class is encoded with LSTM_C into 90 soft tokens.

PT-Hard-None-LSTM_C-TCl loss: 0.8951 BLEU: 0.0596 crystalBLEU: 0.0289

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1066 | 882 (82.74%) | 59 (5.53%) | 58 (5.44%) | 0 (0.00%) |
| bazaarvoice-ostrich | 3.98% | 5.63% | 3.17% | 5.63% | 1206 | 1002 (83.08%) | 29 (2.40%) | 20 (1.66%) | 5 (0.41%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2803 | 2580 (92.04%) | 349 (12.45%) | 282 (10.06%) | 6 (0.21%) |
| flipkart-incubator-Lyrics | 0.00% | 0.00% | 0.00% | 0.00% | 1373 | 970 (70.65%) | 75 (5.46%) | 54 (3.93%) | 0 (0.00%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 831 | 716 (86.16%) | 34 (4.09%) | 34 (4.09%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2173 | 1877 (86.38%) | 81 (3.73%) | 81 (3.73%) | 2 (0.09%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2626 | 2506 (95.43%) | 72 (2.74%) | 70 (2.67%) | 3 (0.11%) |
| shroffk-phoebus | 0.39% | 1.15% | 0.76% | 1.07% | 3222 | 2460 (76.35%) | 157 (4.87%) | 94 (2.92%) | 15 (0.47%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1443 | 1369 (94.87%) | 18 (1.25%) | 11 (0.76%) | 1 (0.07%) |
| nhl-dflib | 3.23% | 2.31% | 3.19% | 2.11% | 1509 | 1228 (81.38%) | 49 (3.25%) | 34 (2.25%) | 4 (0.27%) |
| monarch-initiative-phenol | 15.76% | 13.54% | 13.61% | 14.42% | 1274 | 1017 (79.83%) | 57 (4.47%) | 40 (3.14%) | 6 (0.47%) |
| Total | 1.77% | 2.39% | 1.76% | 2.35% | 19526 | 16607 (85.05%) | 980 (5.02%) | 778 (3.98%) | 42 (0.22%) |

Table A.12: Results for PT-Hard-None-LSTM_C-TCl, where the test class is encoded with LSTM_C into 150 soft tokens.

PT-Hard-None-LSTM_C-TCl loss: 1.1903 BLEU: 0.0601 crystalBLEU: 0.0292

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1093 | 884 (80.88%) | 50 (4.57%) | 49 (4.48%) | 0 (0.00%) |
| bazaarvoice-ostrich | 3.98% | 5.63% | 3.17% | 5.63% | 1195 | 1006 (84.18%) | 20 (1.67%) | 14 (1.17%) | 3 (0.25%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2772 | 2510 (90.55%) | 301 (10.86%) | 235 (8.48%) | 1 (0.04%) |
| flipkart-incubator-Lyrics | 3.09% | 1.21% | 4.76% | 0.83% | 1356 | 962 (70.94%) | 60 (4.42%) | 37 (2.73%) | 1 (0.07%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 856 | 700 (81.78%) | 23 (2.69%) | 23 (2.69%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2164 | 1905 (88.03%) | 70 (3.23%) | 69 (3.19%) | 0 (0.00%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2597 | 2479 (95.46%) | 74 (2.85%) | 70 (2.70%) | 5 (0.19%) |
| shroffk-phoebus | 0.39% | 1.15% | 0.76% | 1.07% | 3280 | 2462 (75.06%) | 162 (4.94%) | 96 (2.93%) | 21 (0.64%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1442 | 1364 (94.59%) | 33 (2.29%) | 20 (1.39%) | 1 (0.07%) |
| nhl-dflib | 0.00% | 0.00% | 0.00% | 0.00% | 1543 | 1287 (83.41%) | 43 (2.79%) | 30 (1.94%) | 0 (0.00%) |
| monarch-initiative-phenol | 15.22% | 10.64% | 12.43% | 11.52% | 1304 | 1051 (80.60%) | 62 (4.75%) | 44 (3.37%) | 5 (0.38%) |
| Total | 1.68% | 2.02% | 1.63% | 1.97% | 19602 | 16610 (84.74%) | 898 (4.58%) | 687 (3.50%) | 37 (0.19%) |

Table A.13: Results for PT-Hard-None-LSTM_C-TCl, where the test class is encoded with LSTM_C into 188 soft tokens.

### A.3.3 Sum

PT-Hard-None-Sum-TCl loss: 0.8939 BLEU: 0.0532 crystalBLEU: 0.0272

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1128 | 915 (81.12%) | 52 (4.61%) | 52 (4.61%) | 1 (0.09%) |
| bazaarvoice-ostrich | 3.98% | 5.63% | 3.17% | 5.63% | 1278 | 1132 (88.58%) | 23 (1.80%) | 16 (1.25%) | 6 (0.47%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2997 | 2710 (90.42%) | 342 (11.41%) | 297 (9.91%) | 1 (0.03%) |
| flipkart-incubator-Lyrics | 3.47% | 1.36% | 5.56% | 0.97% | 1467 | 996 (67.89%) | 48 (3.27%) | 28 (1.91%) | 3 (0.20%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 911 | 744 (81.67%) | 23 (2.52%) | 22 (2.41%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 1.53% | 1.58% | 0.74% | 1.64% | 2186 | 1903 (87.05%) | 85 (3.89%) | 81 (3.71%) | 1 (0.05%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2741 | 2605 (95.04%) | 78 (2.85%) | 73 (2.66%) | 7 (0.26%) |
| shroffk-phoebus | 0.47% | 1.32% | 0.76% | 1.07% | 3294 | 2476 (75.17%) | 156 (4.74%) | 83 (2.52%) | 18 (0.55%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1543 | 1424 (92.29%) | 25 (1.62%) | 17 (1.10%) | 1 (0.06%) |
| nhl-dflib | 3.23% | 2.31% | 3.19% | 2.11% | 1582 | 1357 (85.78%) | 32 (2.02%) | 26 (1.64%) | 4 (0.25%) |
| monarch-initiative-phenol | 10.05% | 6.00% | 9.47% | 5.72% | 1454 | 1175 (80.81%) | 62 (4.26%) | 44 (3.03%) | 3 (0.21%) |
| Total | 1.85% | 1.93% | 1.82% | 1.77% | 20581 | 17437 (84.72%) | 926 (4.50%) | 739 (3.59%) | 45 (0.22%) |

Table A.14: Results for PT-Hard-None-Sum-TCl, where the test class is encoded with Sum into 1 soft token.

PT-Hard-None-Sum-TCl loss: 0.8978 BLEU: 0.0560 crystalBLEU: 0.0291

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1209 | 995 (82.30%) | 78 (6.45%) | 73 (6.04%) | 0 (0.00%) |
| bazaarvoice-ostrich | 3.98% | 5.63% | 3.17% | 5.63% | 1290 | 1110 (86.05%) | 41 (3.18%) | 30 (2.33%) | 7 (0.54%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2950 | 2635 (89.32%) | 318 (10.78%) | 249 (8.44%) | 16 (0.54%) |
| flipkart-incubator-Lyrics | 3.09% | 1.21% | 4.76% | 0.83% | 1287 | 910 (70.71%) | 43 (3.34%) | 25 (1.94%) | 2 (0.16%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 844 | 716 (84.83%) | 25 (2.96%) | 24 (2.84%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2236 | 2011 (89.94%) | 119 (5.32%) | 116 (5.19%) | 1 (0.04%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2760 | 2632 (95.36%) | 84 (3.04%) | 78 (2.83%) | 5 (0.18%) |
| shroffk-phoebus | 0.79% | 1.97% | 0.38% | 1.97% | 3284 | 2496 (76.00%) | 173 (5.27%) | 92 (2.80%) | 22 (0.67%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1472 | 1407 (95.58%) | 44 (2.99%) | 38 (2.58%) | 4 (0.27%) |
| nhl-dflib | 3.23% | 2.31% | 3.19% | 2.11% | 1545 | 1315 (85.11%) | 43 (2.78%) | 33 (2.14%) | 4 (0.26%) |
| monarch-initiative-phenol | 15.22% | 10.64% | 12.43% | 11.52% | 1421 | 1140 (80.23%) | 64 (4.50%) | 49 (3.45%) | 3 (0.21%) |
| Total | 1.95% | 2.31% | 1.79% | 2.24% | 20298 | 17367 (85.56%) | 1032 (5.08%) | 807 (3.98%) | 64 (0.32%) |

Table A.15: Results for PT-Hard-None-Sum-TCl, where the test class is encoded with Sum into 10 soft tokens.

PT-Hard-None-Sum_C-TCl loss: 0.8939 BLEU: 0.0604 crystalBLEU: 0.0292

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1097 | 886 (80.77%) | 67 (6.11%) | 67 (6.11%) | 1 (0.09%) |
| bazaarvoice-ostrich | 10.62% | 11.27% | 7.94% | 9.51% | 1341 | 1127 (84.04%) | 36 (2.68%) | 28 (2.09%) | 6 (0.45%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2851 | 2632 (92.32%) | 305 (10.70%) | 252 (8.84%) | 9 (0.32%) |
| flipkart-incubator-Lyrics | 1.93% | 0.76% | 1.59% | 0.28% | 1420 | 1052 (74.08%) | 44 (3.10%) | 27 (1.90%) | 3 (0.21%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 894 | 771 (86.24%) | 39 (4.36%) | 38 (4.25%) | 1 (0.11%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2308 | 1981 (85.83%) | 88 (3.81%) | 88 (3.81%) | 3 (0.13%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2758 | 2622 (95.07%) | 111 (4.02%) | 107 (3.88%) | 2 (0.07%) |
| shroffk-phoebus | 0.47% | 1.32% | 0.76% | 1.07% | 3275 | 2467 (75.33%) | 153 (4.67%) | 89 (2.72%) | 15 (0.46%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1499 | 1415 (94.40%) | 23 (1.53%) | 20 (1.33%) | 1 (0.07%) |
| nhl-dflib | 3.23% | 2.31% | 3.19% | 2.11% | 1539 | 1278 (83.04%) | 40 (2.60%) | 33 (2.14%) | 2 (0.13%) |
| monarch-initiative-phenol | 15.22% | 10.64% | 12.43% | 11.52% | 1334 | 1067 (79.99%) | 61 (4.57%) | 45 (3.37%) | 4 (0.30%) |
| Total | 2.08% | 2.72% | 1.95% | 2.46% | 20316 | 17298 (85.14%) | 967 (4.76%) | 794 (3.91%) | 47 (0.23%) |

Table A.16: Results for PT-Hard-None-Sum_C-TCl, where the test class is encoded with Sum into 30 soft tokens.

PT-Hard-None-Sum_C-TCl loss: 0.8922 BLEU: 0.0608 crystalBLEU: 0.0296

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 0.00% | 0.00% | 0.00% | 0.00% | 1107 | 912 (82.38%) | 69 (6.23%) | 69 (6.23%) | 0 (0.00%) |
| bazaarvoice-ostrich | 10.62% | 11.27% | 7.14% | 9.15% | 1314 | 1120 (85.24%) | 34 (2.59%) | 26 (1.98%) | 7 (0.53%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2782 | 2535 (91.12%) | 336 (12.08%) | 279 (10.03%) | 6 (0.22%) |
| flipkart-incubator-Lyrics | 1.93% | 0.76% | 1.59% | 0.28% | 1404 | 1003 (71.44%) | 76 (5.41%) | 45 (3.21%) | 1 (0.07%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 880 | 752 (85.45%) | 21 (2.39%) | 20 (2.27%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2187 | 1916 (87.61%) | 71 (3.25%) | 70 (3.20%) | 2 (0.09%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2677 | 2549 (95.22%) | 118 (4.41%) | 114 (4.26%) | 8 (0.30%) |
| shroffk-phoebus | 0.71% | 1.81% | 0.76% | 1.64% | 3180 | 2386 (75.03%) | 143 (4.50%) | 86 (2.70%) | 19 (0.60%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1534 | 1465 (95.50%) | 31 (2.02%) | 27 (1.76%) | 3 (0.20%) |
| nhl-dflib | 0.81% | 4.00% | 1.20% | 4.00% | 1497 | 1290 (86.17%) | 52 (3.47%) | 35 (2.34%) | 4 (0.27%) |
| monarch-initiative-phenol | 15.22% | 10.64% | 12.43% | 11.52% | 1287 | 1036 (80.50%) | 72 (5.59%) | 47 (3.65%) | 2 (0.16%) |
| Total | 1.76% | 2.78% | 1.50% | 2.53% | 19849 | 16964 (85.47%) | 1023 (5.15%) | 818 (4.12%) | 52 (0.26%) |

Table A.17: Results for PT-Hard-None-Sum_C-TCl, where the test class is encoded with Sum into 90 soft tokens.

PT-Hard-None-Sum_C-TCl loss: 0.8929 BLEU: 0.0597 crystalBLEU: 0.0292

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 0.00% | 0.00% | 0.00% | 0.00% | 1085 | 865 (79.72%) | 52 (4.79%) | 51 (4.70%) | 1 (0.09%) |
| bazaarvoice-ostrich | 1.33% | 2.82% | 1.59% | 2.82% | 1204 | 991 (82.31%) | 31 (2.57%) | 22 (1.83%) | 5 (0.42%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 2825 | 2554 (90.41%) | 342 (12.11%) | 283 (10.02%) | 6 (0.21%) |
| flipkart-incubator-Lyrics | 3.47% | 1.36% | 5.56% | 0.97% | 1385 | 974 (70.32%) | 69 (4.98%) | 44 (3.18%) | 2 (0.14%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 861 | 725 (84.20%) | 31 (3.60%) | 30 (3.48%) | 1 (0.12%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2155 | 1894 (87.89%) | 82 (3.81%) | 79 (3.67%) | 2 (0.09%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2587 | 2448 (94.63%) | 77 (2.98%) | 73 (2.82%) | 5 (0.19%) |
| shroffk-phoebus | 0.39% | 1.15% | 0.63% | 0.99% | 3208 | 2438 (76.00%) | 157 (4.89%) | 101 (3.15%) | 24 (0.75%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1435 | 1368 (95.33%) | 36 (2.51%) | 27 (1.88%) | 3 (0.21%) |
| nhl-dflib | 3.50% | 3.64% | 3.59% | 3.44% | 1526 | 1272 (83.36%) | 43 (2.82%) | 31 (2.03%) | 3 (0.20%) |
| monarch-initiative-phenol | 15.22% | 10.64% | 12.43% | 11.52% | 1283 | 1024 (79.81%) | 63 (4.91%) | 43 (3.35%) | 10 (0.78%) |
| Total | 1.58% | 1.97% | 1.59% | 1.90% | 19554 | 16553 (84.65%) | 983 (5.03%) | 784 (4.01%) | 62 (0.32%) |

Table A.18: Results for PT-Hard-None-Sum_C-TCl, where the test class is encoded with Sum into 150 soft tokens.

PT-Hard-None-Sum_C-TCl loss: 0.8937 BLEU: 0.0611 crystalBLEU: 0.0296

| Repository | Coverage | | | | Counts | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | MiLC | MaLC | MiBC | MaBC | Unique | Parsable | Compilable | Executable | Correct |
| criteo-garmadon | 2.85% | 1.54% | 3.72% | 1.40% | 1137 | 896 (78.80%) | 51 (4.49%) | 50 (4.40%) | 0 (0.00%) |
| bazaarvoice-ostrich | 10.62% | 11.27% | 7.14% | 9.15% | 1288 | 1121 (87.03%) | 39 (3.03%) | 25 (1.94%) | 6 (0.47%) |
| adorsys-XS2A-Sandbox | 0.00% | 0.00% | 0.00% | 0.00% | 3016 | 2755 (91.35%) | 458 (15.19%) | 400 (13.26%) | 5 (0.17%) |
| flipkart-incubator-Lyrics | 0.00% | 0.00% | 0.00% | 0.00% | 1438 | 972 (67.59%) | 69 (4.80%) | 48 (3.34%) | 0 (0.00%) |
| seedstack-seed | 0.00% | 0.00% | 0.00% | 0.00% | 908 | 748 (82.38%) | 31 (3.41%) | 31 (3.41%) | 0 (0.00%) |
| intuit-QuickBooks-V3-Java-SDK | 0.17% | 0.62% | 0.25% | 0.23% | 2182 | 1871 (85.75%) | 82 (3.76%) | 80 (3.67%) | 1 (0.05%) |
| gooddata-gooddata-java | 0.00% | 0.00% | 0.00% | 0.00% | 2616 | 2506 (95.80%) | 106 (4.05%) | 105 (4.01%) | 4 (0.15%) |
| shroffk-phoebus | 0.39% | 1.15% | 0.63% | 0.99% | 3312 | 2495 (75.33%) | 166 (5.01%) | 111 (3.35%) | 18 (0.54%) |
| messaginghub-pooled-jms | 2.67% | 1.46% | 1.53% | 0.94% | 1495 | 1403 (93.85%) | 27 (1.81%) | 24 (1.61%) | 2 (0.13%) |
| nhl-dflib | 3.23% | 2.31% | 3.19% | 2.11% | 1541 | 1259 (81.70%) | 40 (2.60%) | 28 (1.82%) | 3 (0.19%) |
| monarch-initiative-phenol | 15.22% | 10.64% | 12.43% | 11.52% | 1379 | 1121 (81.29%) | 79 (5.73%) | 63 (4.57%) | 9 (0.65%) |
| Total | 1.98% | 2.64% | 1.82% | 2.40% | 20312 | 17147 (84.42%) | 1148 (5.65%) | 965 (4.75%) | 48 (0.24%) |

Table A.19: Results for PT-Hard-None-Sum_C-TCl, where the test class is encoded with Sum into 188 soft tokens.

| Name | Training Algorithm | Context | | Metrics | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Test Class (TCl) | Focal Class (FCl) | MiLC | Correct | Loss | crystalBLEU | BLEU |
| Evosuite | - | - | - | 37.78% | - | - | - | - |
| FT-Hard-All | Fine-tuning | Hard | Hard | 13.55% | 2.81% | 0.6743 | 0.0473 | 0.0808 |
| ZS-Hard-All | Zero-Shot | Hard | Hard | 3.07% | 0.29% | 1.0330 | 0.0237 | 0.0509 |
| PT-Hard-All | Prefix Tuning | Hard | Hard | 9.76% | 1.16% | 0.6825 | 0.0355 | 0.0677 |
| PT-Hard-TCl | Prefix Tuning | Hard | - | 5.48% | 0.82% | 0.6887 | 0.0378 | 0.0691 |
| PT-Hard-FCl | Prefix Tuning | - | Hard | 3.28% | 0.73% | 0.8091 | 0.0323 | 0.0647 |
| PT-Hard-None | Prefix Tuning | - | - | 1.79% | 0.12% | 0.8766 | 0.0283 | 0.0550 |
| PT-Hard-None-BERT-TCl-1 | Prefix Tuning | BERT ($l = 1$) | - | 1.79% | 0.19% | 0.8944 | 0.0257 | 0.0557 |
| PT-Hard-None-BERT-TCl-10 | Prefix Tuning | BERT ($l = 10$) | - | 1.82% | 0.19% | 0.8937 | 0.0285 | 0.0586 |
| PT-Hard-None-BERT_C-TCl-30 | Prefix Tuning | BERT_C ($l = 30$) | - | 2.03% | 0.23% | 0.9004 | 0.0304 | 0.0611 |
| PT-Hard-None-BERT_C-TCl-90 | Prefix Tuning | BERT_C ($l = 90$) | - | 1.92% | 0.28% | 0.9051 | 0.0328 | 0.0654 |
| PT-Hard-None-BERT_C-TCl-150 | Prefix Tuning | BERT_C ($l = 150$) | - | 1.89% | 0.23% | 1.1792 | 0.0306 | 0.0627 |
| PT-Hard-None-BERT_C-TCl-188 | Prefix Tuning | BERT_C ($l = 188$) | - | 1.79% | 0.38% | 1.1810 | 0.0312 | 0.0632 |
| PT-Hard-None-LSTM-TCl-1 | Prefix Tuning | LSTM ($l = 1$) | - | 1.49% | 0.18% | 0.8957 | 0.0281 | 0.0587 |
| PT-Hard-None-LSTM-TCl-10 | Prefix Tuning | LSTM ($l = 10$) | - | 1.81% | 0.22% | 0.8908 | 0.0305 | 0.0618 |
| PT-Hard-None-LSTM_C-TCl-30 | Prefix Tuning | LSTM_C ($l = 30$) | - | 1.55% | 0.24% | 0.8964 | 0.0322 | 0.0644 |
| PT-Hard-None-LSTM_C-TCl-90 | Prefix Tuning | LSTM_C ($l = 90$) | - | 2.03% | 0.27% | 0.8941 | 0.0295 | 0.0608 |
| PT-Hard-None-LSTM_C-TCl-150 | Prefix Tuning | LSTM_C ($l = 150$) | - | 1.77% | 0.22% | 0.8951 | 0.0289 | 0.0596 |
| PT-Hard-None-LSTM_C-TCl-188 | Prefix Tuning | LSTM_C ($l = 188$) | - | 1.68% | 0.19% | 1.1903 | 0.0292 | 0.0601 |
| PT-Hard-None-Sum-TCl-1 | Prefix Tuning | Sum ($l = 1$) | - | 1.85% | 0.22% | 0.8939 | 0.0272 | 0.0532 |
| PT-Hard-None-Sum-TCl-10 | Prefix Tuning | Sum ($l = 10$) | - | 1.95% | 0.32% | 0.8978 | 0.0291 | 0.0560 |
| PT-Hard-None-Sum_C-TCl-30 | Prefix Tuning | Sum_C ($l = 30$) | - | 2.08% | 0.23% | 0.8939 | 0.0292 | 0.0604 |
| PT-Hard-None-Sum_C-TCl-90 | Prefix Tuning | Sum_C ($l = 90$) | - | 1.76% | 0.26% | 0.8922 | 0.0296 | 0.0608 |
| PT-Hard-None-Sum_C-TCl-150 | Prefix Tuning | Sum_C ($l = 150$) | - | 1.58% | 0.32% | 0.8929 | 0.0292 | 0.0597 |
| PT-Hard-None-Sum_C-TCl-188 | Prefix Tuning | Sum_C ($l = 188$) | - | 1.98% | 0.24% | 0.8937 | 0.0296 | 0.0611 |

Table A.20: Summary of the results.

| Training Method | MiLC | Executable | Correct | Loss | crystalBLEU | BLEU |
|---|---|---|---|---|---|---|
| Fine-tuning | 13.55% | 6.42% | 2.81% | 0.6743 | 0.0473 | 0.0808 |
| Prefix Tuning | 9.76% | 6.93% | 1.16% | 0.6825 | 0.0355 | 0.0677 |
| Zero-Shot | 3.07% | 9.43% | 0.29% | 1.0330 | 0.0237 | 0.0509 |

Table A.21: Summary of the results.

# Bibliography


[1] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics.

[2] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti. A3Test: Assertion-Augmented Automated Test Case Generation, Feb. 2023. arXiv:2302.10352 [cs].

[3] M. Allamanis. The Adverse Effects of Code Duplication in Machine Learning Models of Code, Aug. 2019. arXiv:1812.06469 [cs].

[4] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272, Buenos Aires, Argentina, May 2017. IEEE.

[5] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program Synthesis with Large Language Models, Aug. 2021. Number: arXiv:2108.07732 arXiv:2108.07732 [cs].

[6] P. Bareiß, B. Souza, M. d'Amorim, and M. Pradel. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code, June 2022. Number: arXiv:2206.01335 arXiv:2206.01335 [cs].

[7] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman. GPT-Neo: Large scale autoregressive language modeling with meshtensorflow, Oct. 2021.

[8] E. Bogomolov, S. Zhuravlev, E. Spirin, and T. Bryksin. Assessing Project-Level Fine-Tuning of ML4SE Models, June 2022. Number: arXiv:2206.03333 arXiv:2206.03333 [cs].

[9] D. Bowes, T. Hall, J. Petric, T. Shippey, and B. Turhan. How Good Are My Tests? In *2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 9–14, Buenos Aires, Argentina, May 2017. IEEE.

[10] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, and others. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[11] J. Chen, A. Zhang, X. Shi, M. Li, A. Smola, and D. Yang. Parameter-Efficient Fine-Tuning Design Spaces, Jan. 2023. arXiv:2301.01821 [cs].

[12] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, 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. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, 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, July 2021. arXiv:2107.03374 [cs].

[13] M. Cohn. *Succeeding with agile: software development using Scrum*. The Addison-Wesley signature series. Addison-Wesley, Upper Saddle River, NJ, 2010. OCLC: ocn318420978.

[14] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 107–118, Bergamo Italy, Aug. 2015. ACM.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[16] X. Devroey, A. Gambi, J. P. Galeotti, R. Just, F. Kifetew, A. Panichella, and S. Panichella. JUGE: An infrastructure for benchmarking Java unit test generators. *Software Testing, Verification and Reliability*, page e1838, 2022. Publisher: Wiley Online Library.

[17] A. Eghbali and M. Pradel. CrystalBLEU: precisely and efficiently measuring the similarity of code. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[18] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, Nov. 2020. Association for Computational Linguistics.

[19] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, page 416, Szeged, Hungary, 2011. ACM Press.

[20] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study. *ACM Transactions on Software Engineering and Methodology*, 24(4):1–49, Sept. 2015.

[21] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy. The Pile: An 800GB Dataset of Diverse Text for Language Modeling, Dec. 2020. arXiv:2101.00027 [cs].

[22] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software*, 156:312–327, Oct. 2019.

[23] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the 26th Conference on Program Comprehension*, pages 348–351, Gothenburg Sweden, May 2018. ACM.

[24] Y. Hao, H. Song, L. Dong, S. Huang, Z. Chi, W. Wang, S. Ma, and F. Wei. Language Models are General-Purpose Interfaces, June 2022. arXiv:2206.06336 [cs].

[25] J. He and M. Vechev. Controlling Large Language Models to Generate Secure and Vulnerable Code, Feb. 2023. arXiv:2302.05319 [cs].

[26] S. Huang, L. Dong, W. Wang, Y. Hao, S. Singhal, S. Ma, T. Lv, L. Cui, O. K. Mohammed, B. Patra, Q. Liu, K. Aggarwal, Z. Chi, J. Bjorck, V. Chaudhary, S. Som, X. Song, and F. Wei. Language Is Not All You Need: Aligning Perception with Language Models, Mar. 2023. arXiv:2302.14045 [cs].

[27] R. Just, D. Jalali, and M. 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*, pages 437–440, San Jose CA USA, July 2014. ACM.

[28] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th International Conference on Software Engineering, ser. ICSE*, 2023.

[29] B. Lester, R. Al-Rfou, and N. Constant. The Power of Scale for Parameter-Efficient Prompt Tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, Online and Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics.

[30] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online, July 2020. Association for Computational Linguistics.

[31] X. L. Li and P. Liang. Prefix-Tuning: Optimizing Continuous Prompts for Generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online, Aug. 2021. Association for Computational Linguistics.

[32] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Computing Surveys*, 55(9):1–35, Sept. 2023.

[33] X. Liu, K. Ji, Y. Fu, W. L. Tam, Z. Du, Z. Yang, and J. Tang. P-Tuning v2: Prompt Tuning Can Be Comparable to Fine-tuning Universally Across Scales and Tasks, Mar. 2022. Number: arXiv:2110.07602 arXiv:2110.07602 [cs].

[34] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[35] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis, Sept. 2022. arXiv:2203.13474 [cs].

[36] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA '07*, page 815, Montreal, Quebec, Canada, 2007. ACM Press.

[37] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84, Minneapolis, MN, USA, May 2007. IEEE. ISSN: 0270-5257.

[38] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia. On the diffusion of test smells in automatically generated test code: an empirical study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, pages 5–14, Austin Texas, May 2016. ACM.

[39] A. Panichella, F. M. Kifetew, and P. Tonella. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2018.

[40] M. Pradel and S. Chandra. Neural Software Analysis. *Commun. ACM*, 65(1):86–96, Dec. 2021. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[41] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language Models are Unsupervised Multitask Learners. 2019.

[42] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020. Publisher: JMLRORG.

[43] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. Adaptive Test Generation Using a Large Language Model, Feb. 2023. arXiv:2302.06527 [cs].

[44] D. Shrivastava, H. Larochelle, and D. Tarlow. Repository-Level Prompt Generation for Large Language Models of Code, June 2022. Number: arXiv:2206.12839 arXiv:2206.12839 [cs].

[45] T. Tang, J. Li, W. X. Zhao, and J.-R. Wen. Context-Tuning: Learning Contextualized Prompts for Natural Language Generation. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 6340–6354, Gyeongju, Republic of Korea, Oct. 2022. International Committee on Computational Linguistics.

[46] M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy. Methods2Test: A Dataset of Focal Methods Mapped to Test Cases. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, pages 299–303, New York, NY, USA, 2022. Association for Computing Machinery. event-place: Pittsburgh, Pennsylvania.

[47] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan. Unit Test Case Generation with Transformers and Focal Context, May 2021. Number: arXiv:2009.05617 arXiv:2009.05617 [cs].

[48] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, \. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[49] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu. No More Fine-Tuning? An Experimental Evaluation of Prompt Tuning in Code Intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 382–394, New York, NY, USA, 2022. Association for Computing Machinery. event-place: Singapore, Singapore.

[50] 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 *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics.

[51] Z. Wu, S. Wang, J. Gu, R. Hou, Y. Dong, V. Vydiswaran, and H. Ma. IDPG: An Instance-Dependent Prompt Generation Method. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5507–5521, Seattle, United States, July 2022. Association for Computational Linguistics.

[52] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, pages 1–10, New York, NY, USA, 2022. Association for Computing Machinery. event-place: San Diego, CA, USA.

[53] F. Zhang, B. Chen, Y. Zhang, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation, Mar. 2023. arXiv:2303.12570 [cs].

[54] A. Zlotchevski, D. Drain, A. Svyatkovskiy, C. B. Clement, N. Sundaresan, and M. Tufano. Exploring and Evaluating Personalized Models for Code Generation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 1500–1508, New York, NY, USA, 2022. Association for Computing Machinery. event-place: Singapore, Singapore.