Learning Deep Semantics for Test Completion

Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, Milos Gligoric
UT Austin, USA
{pynie,rahulb517,jessy,mooney,gligoric}@utexas.edu

***Abstract*—Writing tests is a time-consuming yet essential task during software development. We propose to leverage recent advances in deep learning for text and code generation to assist developers in writing tests. We formalize the novel task of test completion to automatically complete the next statement in a test method based on the context of prior statements and the code under test. We develop TECO—a deep learning model using code semantics for test completion. The key insight underlying TECO is that predicting the next statement in a test method requires reasoning about code execution, which is hard to do with only syntax-level data that existing code completion models use. TECO extracts and uses six kinds of code semantics data, including the execution result of prior statements and the execution context of the test method. To provide a testbed for this new task, as well as to evaluate TECO, we collect a corpus of 130,934 test methods from 1,270 open-source Java projects. Our results show that TECO achieves an exact-match accuracy of 18, which is 29% higher than the best baseline using syntax-level data only. When measuring functional correctness of generated next statement, TECO can generate runnable code in 29% of the cases compared to 18% obtained by the best baseline. Moreover, TECO is significantly better than prior work on test oracle generation.**

***Index Terms*—test completion, deep neural networks, programming language semantics**

I. INTRODUCTION

Software testing is the most common approach in industry to check the correctness of software. However, manually writing tests is tiresome and time-consuming.

One option is to automatically generate tests. Researchers have proposed a number of techniques in this domain, including fuzz testing [1]–[3], property-based testing [4]–[10], search-based testing [11], [12], combinatorial testing [13], etc. Despite being effective in detecting software bugs, these techniques generate tests with stylistic issues, as test code generated through these techniques rarely resemble manually-written tests [14]–[16] and can be hard to maintain. As a result, these automated techniques end up being used only as supplements to manually-written tests.

Another option is to use machine learning (ML), namely training a model on existing manually-written tests and applying it when writing new tests, which is a plausible methodology supported by the naturalness of software [17], [18]. Advances in deep learning such as recurrent neural networks [19], [20] and large-scale pre-trained transformer models [21]–[24] have led to promising new research in a variety of software engineering tasks, such as code completion [25]–[30] and code summarization [31]–[35]. Code generated with modern models are intelligible to humans, yet we cannot fully rely on them to generate large chunks of meaningful code, or expect them to understand larger project context.

Our goal is to design machine learning approaches to aid developer productivity when *writing tests*. We present a novel task—*test completion*—to help developers write tests faster. Specifically, once a developer starts writing a test method, she can leverage test completion to automatically obtain the next statement in the test code (at any point she desires).

Despite being closely related to code completion [25]–[29], test completion is distinct in that test code has several unique characteristics. First, the method under test provides extra context that can be leveraged when completing a test method. Second, test code follows a different programming style that focuses on exercising the method under test. Specifically, a test method usually consists of a sequence of statements in the following order: prepare inputs to the method under test, execute method under test, and check the results of the execution using assert statements (i.e., test oracles).

We present the first deep learning solution—TECO—that takes into account these unique characteristics of tests.

> TECO *uses code semantics as inputs for novel ML models and performs reranking via test execution.*

Code semantics refers to the information related to test/code execution not available in the syntax-level data (i.e., source code). TECO extracts code semantics (e.g., types of local variables) using software engineering tools and feeds them directly to the model. Once top-k predictions are produced, TECO further ensures the output quality by *executing* the generated statements, and prioritize the runnable and compilable statements over the others.

We design the code semantics used by TECO based on our experience with software analysis in order to best capture the unique characteristics of the test completion task. In total, we consider six different kinds of code semantics that can be grouped to two categories: (1) execution result, including the types of the local variables and whether fields are initialized; (2) execution context, including the setup and teardown methods, the last called method in the test method, and statements in non-test code with similar previous statements.

We implemented TECO to support test methods written in Java. We evaluate TECO on a newly collected corpus consisting of 130,934 test methods with 645,633 statements from 1,270 projects. We release this corpus to the community as a testbed for the test completion task.

We performed extensive evaluations of TECO on this corpus—covering lexical similarity, functional correctness, and downstream application—to show the importance of

combining code semantics with deep learning. We report results comparing the generated statements against the gold manually-written statements using a suite of automatic metrics: exact-match accuracy, top-10 accuracy, BLEU [36], CodeBLEU [37], edit similarity [29], and ROUGE [38]. TECO significantly outperforms baselines that use only syntax-level data on all metrics. We also measure functional correctness by trying to compile and run the generated statements. TECO can produce a runnable next statement 29% of the time, while the figure for the best baseline model is only 18%. Moreover, we also evaluated TECO on the task of test oracle generation [39], [40], which is a downstream application of test completion. TECO achieves an exact-match accuracy of 16, which significantly outperforms the prior state-of-the-art's exact-match accuracy of 9.

The main contributions of this paper include the following:

- **Task**. We propose a novel task, test completion, with the goal to help developers write test methods faster.
- **Idea**. We propose using code semantics and code execution when designing ML models targeting code-related tasks.
- **Model**. We developed TECO, the first transformer model trained on large code semantics data for test completion. Furthermore, TECO performs reranking by execution. The use of code semantics is vital for correctly modeling the execution process in the test methods.
- **Corpus**. We created a large corpus of 130,934 test methods from 1,270 open-source projects. We believe this corpus will also be useful to many other tasks related to testing.
- **Evaluation**. Our extensive evaluation shows that TECO significantly outperforms strong baselines on all automatic metrics, both on test completion and its downstream application: test oracle generation. We also evaluate the functional correctness of generated code by compiling and running the generated statements.

TECO and our corpus are publicly available on GitHub: https://github.com/EngineeringSoftware/teco.

II. TASK

In this section, we more formally describe the test completion task and illustrate the task using an example.

Given an incomplete test method, our goal is to automatically generate the next statement in that test method. We assume that the following inputs are provided to a test completion system: (1) the code under test, which includes both the test method's associated method under test as well as other non-test-method code in the project, (2) the test method signature, (3) prior statements in the incomplete test method (which can be zero or more statements).

We illustrate our task in Fig. 1. The example shows (in the yellow boxes) the method under test, the test method signature, and the prior statements (only one statement in this example), as well as (in the last green box) the next statement that should be generated by a test completion system.

method under test

```
public GMOperation addImage(final File file) {
  if (file == null) {
    throw new IllegalArgumentException(
      "file must be defined"); }
  getCmdArgs().add(file.getPath());
  return this; }
```

test method signature

```
@Test
public void addImage_ThrowsException_WhenFileIsNull()
throws Exception
```

prior statements

```
exception.except(IllegalArgumentException.class);
```

next statement

```
sut.addImage((File) null);
```

Fig. 1: Example of test completion: given the code under test (represented by the method under test), test method signature, and prior statements, the goal is to generate the next statement. Code from `sharneng/gm4java` in class `GMOperationTest`.

We seek to generate statements in the body of the test method, thus the test method signature (including the annotation and the name of test method) are only used as inputs and they are not the prediction target of the test completion task. We also do not consider the context of other already available test methods from the same project when completing a test method, to prevent any model from cheating by copying code from other similar test methods. Our defined test completion task is applicable to the situation when a developer already knows what to test (thus knows the method under test and the test method signature), and wants to complete the next statement at any point when writing the test method, regardless of whether the project has existing tests or not. We also focus on modeling the body of test methods as a sequence of statements, because test methods with control flows (e.g., if statements, loops, and try blocks) are rare; we found less than 10% test methods have control flows in our experiments. Most testing frameworks recommend sequential test method body and provide annotations to replace control flows, for example, `@ParameterizedTest` for replacing loops in JUnit 5 [41].

III. EXTRACTION OF CODE SEMANTICS

In this section, we describe the six kinds of code semantics extracted and used by TECO.

For each kind of code semantics, we design and implement a static analysis algorithm to extract it. Static analysis is the analysis of code without executing it, guided by the grammar and semantics of programming languages. The advantage of using static analysis is that it does not require configuring the runtime environment which can be cumbersome for some projects, and can be applied on partial code (for example, without accessing the dependency libraries of a project, which is needed when executing the code). It is also much faster than executing the code directly, which enables us to collect code semantics on a large corpus of code. However, static

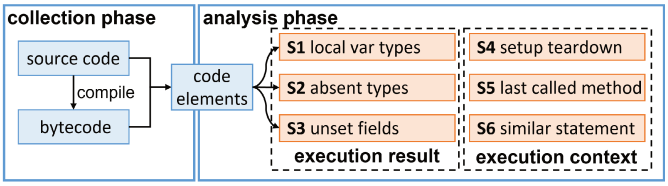

Fig. 2: TECO's workflow of using static analysis to extract code semantics.

analysis can sometimes be inaccurate; for example, when some values are unknown without executing code (e.g., user inputs), static analysis has to over-estimate the analysis results (e.g., assuming both branches of an if statement may be executed). This may not be a problem for TECO as the deep learning model can learn to ignore the inaccurate parts.

Fig. 2 illustrates the general workflow of TECO's static analysis which consists of two phases: given a project, in the collection phase TECO collects a shared set of all code elements (classes, methods, fields), and in the analysis phase TECO extracts each kind of code semantics from the code elements set using a specific algorithm. The code semantics can be organized into two categories based on their content: *execution result* and *execution context*. Execution result includes: (S1) local var types, (S2) absent types, and (S3) unset fields; execution context includes: (S4) setup teardown, (S5) last called method, and (S6) similar statement. In the following paragraphs we describe the collection phase and the analysis phase for each kind of code semantics in more detail.

The collection phase. The goal of this phase is to collect a set of code elements that will be shared by all test methods and all six kinds of code semantics. TECO collects three kinds of code elements: classes, methods, and fields; each method and field should have one class as its parent. For each element, TECO collects its metadata, including name, type, access modifiers, annotations (if any), etc. For each class, TECO records if it is a non-test class, test class, or a class in a dependency. TECO additionally collects the source code and bytecode for all their methods of non-test and test classes. We do not utilize the source code or bytecode in dependency libraries because they are not needed for the analysis.

(S1) local var types data refers to the types of the local variables in the test method. The types are extracted by partially interpreting the bytecode of the test method without considering the values of variables. Note that this is more accurate than reading the local variable table which contains the declared types of local variables; for example, after interpreting the statement `AbstractWComponent comp = new SimpleComponent()`, the type of `comp` is `SimpleComponent`, which is more accurate than its declared type `AbstractWComponent`. This data provides information on what types of test inputs are available to be used in the next statement.

(S2) absent types are the types of the variables that are needed by calling the method under test but have not been prepared in the test method. Types needed by calling the method under test include its parameter types, plus the class under test (the declaring class of the method under test) if the method under test is not a static method. A type is prepared if a local variable or a field of the test class with this type is initialized. This data focuses on what types of test inputs are missing, and thus may likely need to be prepared in the next statement.

(S3) unset fields data refers to the fields of the test class and the class under test that have not been initialized. We deem a field as is initialized if there is any statement for setting the value of the field in the test method, the setup methods, or methods transitively called from the test method or the setup methods (up to 4 jumps, as initializations of the fields of our interest in more in-depth calls are rare). The fields in this data may likely need to be initialized in the next statement.

(S4) setup teardown data refers to the source code of the setup and teardown methods in the test class. When a test framework executes tests, setup methods are executed before the test method to set up the environment (e.g., connection to a database), and teardown methods are executed after the test method to clean up the environment. By providing this context, the test completion system can know what environment is available to use in the test method, and also can avoid duplicating the statements already in setup/teardown methods.

(S5) last called method data is the source code of the last called method in the prior statements, which could be empty if no method has been called yet. This data provides more context on what has been executed in prior statements.

(S6) similar statement data is a statement in the non-test code of the project that has the most similar prior statements context to the prior statements in the incomplete test method. TECO uses the BM25 algorithm [42] to search for similar prior statements. Only 2 prior statements are considered during the search, as increasing the window size leads to much longer search time without improving the quality of the returned similar statement. We expect this data to be similar to the next statement to be predicted.

Implementation. In the collection phase, TECO uses JavaParser [43] to collect source code and ASM [44] to collect bytecode. The collection phase takes 136s per project on average. All analysis algorithms are implemented in Python, with the help of ASM for partially interpreting bytecode (for S1) and scikit-learn [45] for the BM25 algorithm. The analysis phase takes 0.018s–0.247s per test method on average, depending on code semantics data.

Example. Fig. 3 shows two kinds of code semantics (S2 and S4) extracted for the example in Fig. 1 that help the generating the correct next statement. The (S2) absent types data is `File`, because calling the method under test `addImage` requires `GMOperation` (because `addImage` is not a static method) and `File`, but `GMOperation` is already available as a field `sut` in the test class (on line 7). The (S4) setup teardown data is the method `setup` in the test class `GMOperationTest` which initializes the `sut` field (on line 9). Note that the other kinds of code semantics are either empty or not useful for this example, but are useful for some other examples.

```
1 public class GMOperation
2    extends org.im4java.core.GMOperation {
3  public GMOperation addImage(final File file) {...}
4 ...}                                  (S2) absent types
5
6 public class GMOperationTest {
7  GMOperation sut;
8  @Before
9  public void setup() {... sut = new GMOperation(); ...}
10                                      (S4) setup teardown
11 @Test
12 public void addImage_ThrowsException_WhenFileIsNull()
13    throws Exception {
14  exception.except(IllegalArgumentException.class);
15  sut.addImage((File) null);
16 }
17 ...}
```

Fig. 3: Some kinds of code semantics (S2 and S4, highlighted in the first two blue boxes) extracted for the example in Fig. 1, which help TECO generate the correct next statement (highlighted in the last green box).

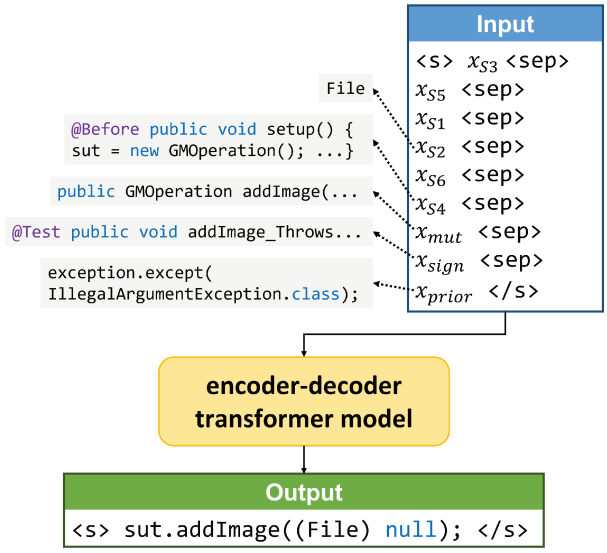

Fig. 4: TECO's model architecture.

IV. TECO'S DEEP LEARNING MODEL

This section describes the deep learning approach that TECO uses to solve the test completion task. Fig. 4 illustrates the overall model architecture: an encoder-decoder transformer model whose input includes both code semantics and syntax-level data and output is the next statement.

A. Encoder-Decoder Transformer Model

Our model is based on the encoder-decoder architecture that considers both input and output as sequences, which has been applied to many sequence generation tasks including code summarization [31]–[33] and code generation [46], [47]. In the context of TECO, the input is the syntax-level data (method under test, test signature, and prior statements) plus the code semantics extracted from the test to be completed (S1-S6), and the output is the next statement.

More formally, TECO is given the test to be completed T and the code under test C as inputs, where C includes the method under test x_{mut}, and T consists of two parts: the test signature x_{sign} and prior statements x_{prior}. The goal is to generate the next statement y. TECO extracts code semantics as described in Section III:

$$x_{S1}, x_{S2}, x_{S3}, x_{S4}, x_{S5}, x_{S6} = \texttt{analysis}(T, C)$$

Each input piece x and output y is a sequence of subtokens, which is obtained by subtokenizing the code or extracted data's string format using the BPE (byte-pair encoding) algorithm [22], [48]. TECO combines the input pieces into a single input sequence by concatenating them with a delimiter $\langle\texttt{sep}\rangle$ (the ordering of the sequences is configurable):

$$x = x_{S3}\ \langle\texttt{sep}\rangle\ x_{S5}\ \langle\texttt{sep}\rangle\ x_{S1}\ \langle\texttt{sep}\rangle\ x_{S2}\ \langle\texttt{sep}\rangle\ x_{S6}\ \langle\texttt{sep}\rangle\ x_{S4}\ \langle\texttt{sep}\rangle\ x_{mut}\ \langle\texttt{sep}\rangle\ x_{sign}\ \langle\texttt{sep}\rangle\ x_{prior}$$

The maximum number of subtokens that TECO can accept, due to the limitation of the underlying model, is 512. If the input sequence is longer than that, TECO truncates the input sequence from the beginning. In our experiments, the number of subtokens in the input sequence ranges from 23 to 2,426 (average: 243.88) and exceeds the limitation of 512 subtokens in 6% of the cases. As a result, more important information should be placed at the end of the input sequence to avoid being truncated. The ordering of code semantics is decided based on our domain knowledge of which kind of data would contain more important information, and we always put the syntax-level data at the end (as they deliver the basic information for the task). Exploring all possible orderings may discover better models but is too computationally expensive. We plan to investigate the impact of the orderings by designing experiments with more affordable costs in the future.

Then, TECO uses a transformer model [21] to learn the conditional probability distribution $P(y|x)$. The model computes this by first using an encoder to encode the input sequence into a deep representation, and then using a decoder which reads the deep representation and generates the output sequence one subtoken at a time:

$$h = \texttt{encoder}(x)$$
$$P(y[i]|y[:i], x) = \texttt{decoder}(y[:i], h), \text{for each } i$$

B. Fine-tuning

Recent work shows that pre-training a large-scale transformer model on a large corpus of code and text and then fine-tuning the model on downstream tasks lead to better performance than training a model from scratch [30], [34], [35], [49]. However, pre-training is only performed on syntax-level data in order to be generalizable to many downstream tasks with different semantics. Prior work shows that large-scale pre-trained models may not perform well on simple tasks of executing the code [50]. This indicates that they learned little about the semantics of execution.

We believe that fine-tuning on code semantics is vital for pre-trained models to perform well on execution-related tasks

such as test completion. During pre-training, the model mainly learns the syntax and grammar of programming languages. If only syntax-level data is used during fine-tuning, the model would have to infer the semantics of execution, which can be very inaccurate because the execution can be complicated and the context provided by the syntax-level data is limited. By contrast, code semantics are extracted using reliable static analysis algorithms in TECO so that the model can directly use such data instead of inferring. Thus, during fine-tuning of TECO, the model learns how to understand and process the additional code semantics in addition to the syntax-level data.

For the pre-trained model, we use CodeT5 [35], which is a large-scale encoder-decoder transformer model pre-trained on a bimodal corpus of code in multiple programming languages (including Java) and text. We fine-tune the model on a corpus for test completion $\mathcal{TC}$ by minimizing the cross-entropy loss:

$$\texttt{loss} = \sum_{x,y \in \mathcal{TC}} -\log P(y|x) = \sum_{\substack{x,y \in \mathcal{TC} \\ i \in [0,|y|)}} -\log P(y[i]|y[:i], x)$$

C. Evaluation

At evaluation time, TECO uses the beam search algorithm with a beam size of 10. Specifically, starting from a special begin-of-sequence subtoken $y[0] = \langle \texttt{s} \rangle$, TECO iteratively runs `decoder` to generate the most likely next subtokens which is appended to the output sequence; only the top 10 sequences with the highest total probability are kept at each step. Each output sequence is completed upon generating a special end-of-sequence subtoken $\langle /\texttt{s} \rangle$. The beam search terminates after generating 10 completed output sequences. As repeating the same subtoken is unlikely, we apply a repetition prevention mechanism that penalizes the probability of generating the same subtoken as the previous subtoken [51].

D. Reranking by Execution

The model can generate plausible outputs by maximizing the generation probability that it learnt during fine-tuning. However, there is no guarantee for the generated statements—subtoken sequences—to be compilable and runnable code. Arguably, generating compilable and runnable code is more important than generating plausible but non-executable code in the test completion task, because it is crucial for developers to run the code and observe the runtime behavior, in order to further improve the codebase.

We propose to use *reranking by test execution* to improve the quality of the generated statements. Specifically, after collecting the top-10 predictions from beam search $\mathcal{Y}$ ranked by their probabilities, TECO checks whether each of them is compilable and runnable. Then, TECO reranks the outputs into $\hat{\mathcal{Y}}$ where one output y_i is ranked higher than another y_j if: (1) y_i is runnable and y_j is not; or (2) both are not runnable, but y_i is compilable and y_j is not; or (3) both have the same runnable and compilable status, and $P(y_i) > P(y_j)$. In this way, generated statements that are compilable and runnable are prioritized over the others.

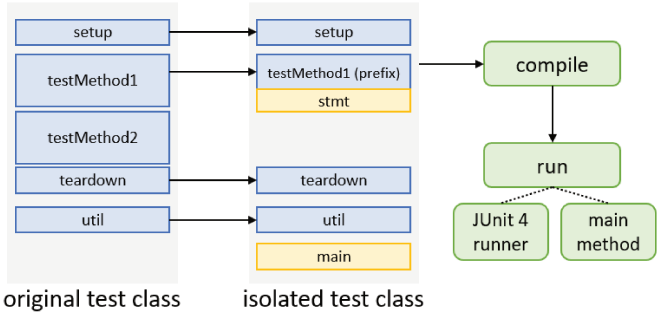

Fig. 5: Procedure of detecting whether a generated statement is compilable and runnable.

TECO detects whether a generated statement is compilable and runnable by putting it in an ad-hoc test class with the required context, isolated from being affected by other test methods in the same project. The procedure is illustrated in Fig. 5, specifically:

1) Create a class with a test method using the signature and prior statements, followed by the generated statement.
2) Extract the other non-test methods from the original test class (including setup, teardown, and utility methods) into the created class.
3) Generate an ad-hoc main method which calls setup methods, the test method, and teardown methods in order.
4) Compile the generated class with all the dependencies specified in the project's build configuration as well as all non-test classes in the project.
5) If the compilation succeeds (at which point the statement is considered to be compilable), execute the compiled class; the statement is considered to be runnable only if there is no exception or assertion failure during the execution. The execution of the class is performed by one of the following:
   a) If the project is using JUnit 4 [52] testing framework, we try to use its command line runner to run the class with the generated statement; the JUnit 4 runner is more robust because it properly utilizes all JUnit features that our ad-hoc main method does not handle, e.g., `@RunWith`.
   b) If the project is not using JUnit 4 or running with JUnit 4 runner failed, we run the ad-hoc main method.

V. CORPUS

As test completion is a new task, we construct a large-scale corpus that can serve as a testbed for our work and future research. We collected data from the same subject projects used by CodeSearchNet [53], which is a large corpus of code and comments that is frequently used in ML+code research [35], [49], [54]. Out of the 4,767 Java projects in CodeSearchNet, we used the 1,535 projects that: (1) use the Maven build system (for the simplicity of data collection; TECO is not limited to any build system); (2) compile successfully, and (3) have a license that permits the use of its data. We collected

TABLE I: Statistics of our corpus. #proj = number of projects; #test = number of test method; #stmt = number of statements; len(test) = average number of tokens in test method; len(MUT) = average number of tokens in method under test.

	#proj	#test	#stmt	len(test)	len(MUT)
all	1,270	130,934	645,633	79.57	40.88
training	1,163	120,521	584,924	79.58	40.61
validation	43	5,413	30,515	73.24	45.09
evaluation	64	5,000	30,194	86.26	42.85

the corpus in Spring 2022. To ensure corpus quality, we try to use the latest stable revision of each project by finding its latest git-tag; but if it does not have any git-tag on or after Jan 1st, 2020, we use its latest revision.

To extract test methods from these projects, we first collected the set of code elements from each project using the same toolchain for the collection phase of TECO's static analysis (Section III). We identified the test methods written in JUnit 4 [52] and JUnit 5 [41] testing frameworks, which are the main frameworks used for writing tests in Java. Specifically, we searched for methods with a test annotation (`@org.junit.Test` or `@org.junit.jupiter.api.Test`) and without an ignored-test annotation (`@org.junit.Ignore` or `@org.junit.jupiter.api.Disabled`). This initial search resulted in 221,666 test methods in all projects.

Then, we further filtered the test methods to ensure corpus quality. We filtered test methods that are badly named (e.g., `test0`; 2,490 cases) or do not follow the required signature of tests (e.g., parameter list is not empty, return type is not void; 1,908 cases). Then, we tried to locate the method under test for each test method, using the following enhanced procedure originally proposed by Waston et al. [39]:

1) If there is only one call to a method, select it (as the method under test);
2) If a class under test can be found by removing "Test" from the test class's name:
   a) If there is only one call to a method declared in class under test, select it;
   b) Select the last method declared in class under test called before the first assertion statement, if any;
3) Select the last method called before the first assertion statement, if any;
4) Select the last method called, if any.

We removed 36,818 test methods for which we could not locate the method under test after this procedure.

We used the line number table to find the bytecode instructions corresponding to each statement, and we removed 633 cases where we could not do this because of multiple statements on the same line. After that, we set size constraints on the data: the test method should have at least 1 statement (filtered 5 cases) and at most 20 statements (filtered 8,222 cases); the method under test should have at most 200 tokens (filtered 9,787 cases); the method under test and the test method together should have at most 400 tokens (filtered 1,288 cases); each statement in the test method should have at most 100 tokens (filtered 1,726 cases).

We also removed several cases that introduce extra overhead during analysis: test methods with if statements, loops, and try blocks, because they entail non-sequential control flow which is not suitable to be modeled by predicting the next statement given prior statements (22,435 cases); and test methods using lambda expressions [55], because they prevent many static analysis algorithms from working (5,420 cases). We plan to lift these limitations in future work.

Lastly, we mask the string literals in the data by replacing them with a common token "STR", similar to prior work on code completion [29]. Although string literals are frequently used in test methods, for example as logging messages, test inputs, or expected outputs, they pose challenges for a pure-deep-learning solution to generate because they have a different style than other parts of the code and can sometimes be very long. Thus, we focus on predicting the next statement with masked string literals, and leave predicting the content of the string literals as future work.

After filtering, we obtained a corpus with 1,270 projects (removed 265 projects because no data was left after filtering), 130,934 test methods, and 645,633 statements.

We follow the same project-level training/validation/evaluation split as CodeSearchNet. Because CodeT5, the pre-trained model that TECO uses in our experiments, also followed the same project-level split, our experiments will not have data leakage issues of evaluating on the data that the model was pre-trained on. Table I shows the statistics of our corpus, where the first row is for the entire corpus, and the other three rows are for each set after the split. Out of the 130,934 test methods, 101,965 (77.88%) are runnable following our procedure described in Section IV-D. Note that with the masking of string literals, some test methods that would originally pass may be considered as "not runnable" in our current corpus (e.g., when the test method compares a variable with a string literal).

VI. EXPERIMENTS SETUP

We assess the performance of TECO by answering the following research questions:

RQ1: What is the performance of TECO on the test completion task and how does it compare to baselines?

RQ2: On the runnable subset of evaluation set, how frequently can TECO predict a compilable and runnable next statement?

RQ3: What is the performance of TECO on test oracle generation, which is a downstream application of the test completion task, and how does it compare to prior work?

RQ4: How does reranking by execution help with more accurately predicting the next statement?

RQ5: How does each kind of code semantics help with more accurately predicting the next statement, and how complementary are different kinds of code semantics?

To answer these questions, we setup an experiment to evaluate TECO and baseline models on our test completion corpus. We train each model on the training and validation sets (validation set is used for tuning hyper-parameters and early stopping), apply the model to predict each statement of each test method in the evaluation set (or subsets of the evaluation set), and measure the quality of the prediction via a number of evaluation metrics, both intrinsically and extrinsically.

All models are trained and evaluated on machines equipped with 4 NVidia 1080-TI GPUs and Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz. We ran each experiment three times with different random seeds and report average values. When comparing models, we conducted statistical significance tests using bootstrap tests [56] with a 95% confidence level.

We next describe the TECO models (Section VI-A) and baseline models (Section VI-B) used in the experiments, the subsets of the evaluation set for computing compilable and runnable metrics and evaluating on the test oracle generation task (Section VI-C), and the evaluation metrics (Section VI-D).

A. TECO *Models*

We run a TECO model that uses all six kinds of code semantics and with reranking by test execution. To study RQ4, we run a TECO-noRr model that uses the same code semantics but does not use reranking. To study RQ5, we run six TECO models with only one kind of code semantics at a time, which we call TECO-ID (e.g., TECO-S1 only uses S1).

B. Baseline Models

We compare our TECO models to the following baseline models that only use syntax-level data.

CodeT5 [35] is a pre-trained encoder-decoder transformer model for code-related tasks, and is built on top of Google's popular T5 framework [24]. CodeT5 was pre-trained on eight commonly used programming languages (including Java) using both mask language modeling and identifier name recovering tasks. We fine-tune TECO models based on CodeT5. As such, we compare to a baseline CodeT5 model that is finetuned on syntax-level data. For completeness, we also compare to a CodeT5-noFt that is only pre-trained and not fine-tuned.

CodeGPT [54] is a decoder-only transformer model built on GPT-2 [22]. We used the `java-adapted` version of it, which is initialized from GPT-2 pre-trained on natural language, and then further pre-trained on a corpus of Java code. Svyatkovskiy et al. [29] used a very similar model (which is not publicly available) for code completion. As CodeGPT tends to generate longer code than a statement (without generating the ⟨/s⟩ subtoken to stop the generation), we slightly modify its decoding algorithm to terminate upon generating the first ';' subtoken for the test completion task.

Test oracle generation is the task of generating the assertion statement given the code under test (including the method under test), test method signature, and prior statements before the assertion statement. When studying this task, we additionally compare to the following two deep learning baseline models for test oracle generation developed in prior work, both of which only use syntax-level data. Following the prior works, we only consider generating the first assertion statement in each test method.

ATLAS [39] is a RNN encoder-decoder model for test oracle generation. We used the "raw model" version of it, i.e., that does not abstract out the identifiers in code.

TOGA [40] is a transformer encoder-only model for classifying the suitability of an assertion statement for an incomplete test method without assertions. It can be used for test oracle generation by first generating a set of assertion statements and then using the model to rank them and select the best one. The model is initialized from CodeBERT [49], which is also pre-trained on the CodeSearchNet corpus [53].

For all baseline models, we use the default hyper-parameters and training configurations recommended by the authors. We train CodeT5 and CodeGPT on the entire training and validation set of our corpus. We train ATLAS and TOGA on a subset of our training and validation set that only predicts the first assertion statement in each test method, which contains 92,567 statements and 3,050 statements, respectively.

C. Subsets of the Evaluation Set

To study the ability of models in predicting a compilable and runnable next statement, we evaluate models on the runnable subset. That is, the subset of the evaluation set where the gold (i.e., developer-written) statement is runnable. We follow the same procedure to check if the gold statement is runnable as described in Section IV-D. Not all gold statements can be successfully executed because of the difficulties in setting up the proper runtime environment, such as missing resources (that may need to be downloaded or generated via other commands), requiring other runtime environments than Java, etc. Our runnable subset contains 25,074 statements (83.04% of all statements in the evaluation set) from 4,223 test methods.

To study the test oracle generation task, we evaluate models on the oracle subset: the subset of the evaluation set where the statement to generate is the first assertion statement in the test method, which contains 4,212 statements. To compute compilable and runnable metrics on the test oracle generation task, we evaluate models on the oracle-runnable subset: the subset of the oracle subset where the gold statement is runnable, which contains 3,540 statements.

D. Evaluation Metrics

(1) Lexical-level metrics: We use the following automatic metrics to measure how close the predicted statements are to the gold statements; these metrics have been frequently used in prior work on code generation and comment generation [31], [33], [57], [58]:

Exact-match accuracy (XM) is the percentage of predicted statements matches exactly with the gold. This metric is the most strict one; each point of improvement directly entails a larger portion of code that is both syntactically and semantically correct, yet it does not take into account paraphrases or give any partial credit.

TABLE II: Results for TECO and baseline models. The best number for each metric is bolded. In each table, numbers marked with the same greek letter prefix are not statistically significantly different.

(a) On the evaluation set.

Model	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	13.57	24.11	38.33	33.88	60.81	62.40
CodeT5-noFt	0.00	0.00	0.00	3.36	1.68	0.02
CodeGPT	12.20	22.67	36.30	31.84	59.09	61.10
TECO	**17.61**	**27.20**	**42.01**	**37.61**	**63.49**	**65.23**

(b) On the runnable subset.

Model	%Compile	%Run	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	54.84	17.62	14.38	25.39	39.26	34.55	61.36	63.15
CodeT5-noFt	0.00	0.00	0.00	0.00	0.00	3.35	1.65	0.03
CodeGPT	53.77	15.13	12.95	24.03	37.19	32.46	59.75	61.90
TECO	**76.22**	**28.63**	**18.96**	**28.40**	**43.15**	**38.45**	**64.12**	**66.09**

(c) On the oracle subset.

Model	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	8.45	24.04	39.03	31.03	66.50	66.63
CodeT5-noFt	$^{\alpha}$0.00	0.00	0.00	1.18	1.86	0.01
CodeGPT	10.56	27.19	40.91	33.33	67.63	67.94
ATLAS	$^{\alpha}$0.21	0.66	21.55	13.39	54.06	50.70
TOGA	9.01	9.01	25.46	24.73	29.60	28.06
TECO	**16.44**	**27.41**	**43.09**	**35.88**	**68.05**	**68.71**

(d) On the oracle-runnable subset.

Model	%Compile	%Run	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	44.45	16.87	8.79	24.37	40.26	32.23	67.08	67.42
CodeT5-noFt	0.00	0.00	$^{\alpha}$0.00	0.00	0.00	1.26	1.87	0.01
CodeGPT	47.39	16.13	10.41	**28.17**	41.56	33.79	67.78	68.43
ATLAS	3.62	1.45	$^{\alpha}$0.23	0.68	21.81	13.56	54.19	50.87
TOGA	25.61	9.37	9.10	9.10	26.51	25.74	31.00	29.38
TECO	**67.93**	**30.29**	**17.37**	27.39	**44.27**	**36.98**	**68.43**	**69.35**

Top-10 accuracy (Acc@10) is the percentage of any top-10 predicted statements matches exactly with the gold. This metric evaluates the use case where the developer can see and select from the top-10 predictions of the model.

BLEU [36] calculates the number of n-grams (consecutive n subtokens) in the prediction that also appear in the gold; specifically, we compute the $1 \sim 4$-grams overlap between the subtokens in the prediction and the subtokens in the gold, averaged between $1 \sim 4$-grams with smoothing method proposed by Lin and Och [59].

CodeBLEU [37] is an improved version of BLEU adapted for code. It is a combination of the traditional BLEU, the BLEU if only considering keywords, syntactical AST match, and semantic data-flow match.

Edit similarity (EditSim) = 1 - Levenshtein edit distance, where the Levenshtein edit distance measures the amount of single-character edits (including insertion, substitution, or deletion) that need to be made to transform the prediction to the gold, normalized by the maximum number of characters in the prediction and the gold. This metric was proposed and used in prior work on code completion [29].

ROUGE [38] measures the overlap between the prediction subtokens and the gold subtokens based on the Longest Common Subsequence statistics, using F1 score.

(2) Functional correctness: The aforementioned metrics only capture the lexical similarity between the prediction against the gold, but the gold statement may not be the only correct solution for competing the next statement. Namely, the prediction can be functionally correct despite being different from the gold statement. To measure the functional correctness, we additionally use the following automatic metrics:

%Compile is the percentage of the predicted statements that are compilable when appended to the incomplete test.

%Run is the percentage of the predicted statements that are compilable and runnable when appended to the incomplete test, without incurring assertion failures or runtime errors.

Note that %Compile and %Run are over-estimations of the functional correctness, as they do not consider whether the underlying logic of the code is meaningful. That said, most functional correctness errors relevant to tests, such as generating the wrong expected outputs, can be captured by the %Run metric. Prior work has used a similar methodology to

TABLE III: Results for TECO without and with reranking by execution. The best number for each metric is bolded. The differences between models for each metric are statistical significant.

(a) On the evaluation set.

Model	XM	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	13.57	38.33	33.88	60.81	62.40
TECO-noRr	15.25	40.84	36.34	62.92	64.71
TECO	**17.61**	**42.01**	**37.61**	**63.49**	**65.23**

(b) On the runnable subset.

Model	%Compile	%Run	XM	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	54.84	17.62	14.38	39.26	34.55	61.36	63.15
TECO-noRr	60.80	19.49	15.99	41.64	36.82	63.38	65.42
TECO	**76.22**	**28.63**	**18.96**	**43.15**	**38.45**	**64.12**	**66.09**

(c) On the oracle subset.

Model	XM	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	8.45	39.03	31.03	66.50	66.63
TECO-noRr	9.92	40.81	32.90	67.32	67.92
TECO	**16.44**	**43.09**	**35.88**	**68.05**	**68.71**

(d) On the oracle-runnable subset.

Model	%Compile	%Run	XM	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	44.45	16.87	8.79	40.26	32.23	67.08	67.42
TECO-noRr	48.13	18.45	9.62	41.55	33.44	67.57	68.41
TECO	**67.93**	**30.29**	**17.37**	**44.27**	**36.98**	**68.43**	**69.35**

evaluate the functional correctness of text-to-code transduction by running generated code with test cases [30], which was performed on a rather small dataset because of the difficulty in collecting manual labelled data. Thanks to the executable nature of tests, we are able to design the two automatic functional correctness metrics for a large corpus.

VII. RESULTS

A. RQ1: Performance of TECO vs. Baseline Models

Table IIa shows the results of TECO and baseline models on solving the test completion task. Our model TECO significantly outperforms all baseline models on all automatic metrics. TECO achieves 17.61 exact-match accuracy, which is 29% higher than the best baseline model, CodeT5's 13.57. This indicates that using code semantics and reranking by execution can greatly improve deep learning model's performance on test completion.

The non-fine-tuned baseline model, CodeT5-noFt, is not capable of solving test completion task. This is because the model is optimized to solve different tasks during pre-training and does not have the domain knowledge of the input-output format of the test completion task.

CodeGPT has shown to be effective on the task of code completion [29], [54], where the primary goal is to continue generating code similar to the context code. However, it performs slightly worse than the encoder-decoder baseline CodeT5 on test completion, because the task requires generating statement in the test method which has different style than the method under test in the provided context.

B. RQ2: Functional Correctness

Table IIb shows the results of TECO and baseline models on the runnable subset, with %Compile and %Run metrics that measure the functional correctness of the generated statements. Our model, TECO can generate runnable statements for 28.63% of the time, and compilable statements for 76.22% of the time, much higher than the best baseline model's 17.62% and 54.84%. On this runnable subset, TECO also outperforms all baseline models on other metrics measuring lexical similarity.

The other two baseline models, CodeT5-noFt and CodeGPT, fail to generate any compilable or runnable statements. After closer inspection, we found that CodeT5-noFt always generate broken non-code outputs, as it is not fine-tuned to process the inputs; and CodeGPT always generate code that is not a valid statement in Java, e.g., code that starts with a method signature.

C. RQ3: Performance on Test Oracle Generation

Tables IIc and IId show the results of the downstream application of test oracle generation, on the oracle subset and the oracle-runnable subset, respectively. TECO significantly improves the exact-match accuracy on this task by a large margin (by 82%), from 9.01 for the prior state-of-the-art, TOGA, to ours 16.44. Note that TOGA's exact-match accuracy is on-par with CodeT5, the model that TECO is fine-tuned from, which confirms that TECO's improvements primarily come from using code semantics and reranking by execution.

TOGA is the strongest prior model on this task in terms of exact-match accuracy. However, it is worse than the CodeT5

baseline model on other metrics that consider partial matches. This is because TOGA is a classification model that ranks a set of assertion statement candidates generated using heuristics, and when the gold statement is not in the set, the model fails to correctly rank a sub-optimal candidate.

D. RQ4: Improvements from Reranking by Execution

Table III shows the results of TECO-noRr (the top-10 accuracy for TECO-noRr is always the same as TECO, because the reranking is performed on top-10 predictions, thus we did not include this metric in the table).

Comparing TECO with TECO-noRr on the evaluation set (Table IIIa), reranking by execution alone contributes to 2 points in exact-match accuracy. However, the improvements over other similarity metrics, which take into account partial matches, are smaller. This indicates that reranking by execution is effective in prioritizing the exact correct generated statement than other non-runnable candidates most of the times, but in a few cases it may prioritize runnable candidates that are less similar to the gold statement than the original top-1. TECO-noRr still significantly outperforms CodeT5 on all metrics. On the runnable subset (Table IIIb), TECO improves both %Compile and %Run over TECO-noRr by large margins, which shows that reranking by execution is an effective strategy for improving the quality of generated statements.

Reranking by execution ended up being very important for improving performance on the task of test oracle generation, as shown on the oracle subset (Table IIIc) and the oracle-runnable subset (Table IIId). For example, TECO outperforms TECO-noRr by 6–8 points in exact-match accuracy and 12 points in %Run. This is because logical errors in assertion statements can be easily found by execution (e.g., generating the wrong expected value will cause an assertion to fail).

E. RQ5: Comparisons of Code Semantics

Tables IVa and IVb show the results of the TECO models with only one kind of code semantics, comparing with the strongest baseline model CodeT5, on the full evaluation set and the oracle subset, respectively. We did not perform statistical significance tests for the results here as the performances of the models are too close. Each model outperforms CodeT5 on at least one metric, meaning that each code semantics provides some information useful for test completion. In Table IVa, TECO-S2 (absent types) is the best model in terms of BLEU, CodeBLEU, EditSim and ROUGE metrics, and TECO-S4 (setup teardown) is the best model in terms of exact-match accuracy and top-10 accuracy, which indicates that these two kinds of code semantics are relatively more important than others. Interestingly, in Table IVb, the models that achieved the best performance among single-data models changed: TECO-S3 (unset fields) is the best model in terms of BLEU, EditSim, and ROUGE, and TECO-S6 (similar statement) is the best model in terms of exact-match accuracy, top-10 accuracy, and CodeBLEU. Thus, different kinds of code semantics provide complementary information for test completion.

TABLE IV: Results for TECO models with only one kind of code semantics on the evaluation set. The best number for each metric is bolded.

(a) On the evaluation set.

Model	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	13.57	24.11	38.33	33.88	60.81	62.40
TECO-S1	13.88	24.93	39.12	34.66	61.58	63.51
TECO-S2	14.06	25.11	**39.56**	**35.17**	**62.20**	**63.92**
TECO-S3	14.04	24.40	38.81	34.24	61.21	62.87
TECO-S4	**14.44**	**25.55**	39.39	35.00	61.63	63.40
TECO-S5	14.05	24.78	38.74	34.34	61.26	63.00
TECO-S6	14.13	24.74	38.70	34.36	60.86	62.52

(b) On the oracle subset.

Model	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
CodeT5	8.45	24.04	39.03	31.03	66.50	66.63
TECO-S1	8.86	24.37	38.03	29.93	65.59	65.95
TECO-S2	8.23	23.69	38.13	30.27	65.52	65.98
TECO-S3	8.72	23.57	**39.90**	31.96	**67.20**	**67.44**
TECO-S4	8.14	23.84	38.54	30.77	65.59	65.92
TECO-S5	8.43	24.10	38.67	30.85	66.17	66.23
TECO-S6	**9.81**	**25.47**	39.88	**32.20**	66.89	66.97

VIII. LIMITATIONS AND FUTURE WORK

We discuss several limitations of our work and the future work inspired by those limitations.

Usability. We envision our models being integrated into an IDE. At any point, a user would be able to see top-k results from our models and potentially decide to use one of the suggestions. This is similar to email completion that has recently been integrated into several popular web-based email clients, e.g., GMail.

Structured representation. Currently we do not considering using any structured representation of code, e.g., abstract syntax trees (ASTs). Such a representation could enhance performance of our models and enable a quick check of validity of generated code. We leave this for future work.

Test-Driven Development (TDD). We assume that code under test is written before tests when defining the test completion task, which is the opposite order of TDD. Future work could explore the mirror task of code completion with a test method context that is applicable to projects adopting TDD.

Testing frameworks. We focused on tests written in the JUnit style. Although other testing frameworks are available (e.g., TestNG), JUnit is the most popular among Java projects.

Large language models for code. Recent large language models that scale up to billions of parameters create new state-of-the-art for many code-related tasks [30], [60], [61]. However, these models do not perform well on simple code execution tasks [50]. Incorporating code semantics into large language models for code is a promising direction which we leave as future work.

IX. Related Work

Automated test generation. Existing automatic test generation work includes fuzz/random testing [1]–[3], property-based testing [4]–[10], search-based testing [11], [12], and combinatorial testing [13]. The typical goal in automated test generation techniques, e.g., Randoop [1] and EvoSuite [12], is to achieve high code coverage of the code under test by generating a large amount of tests, either randomly or systematically. However, the generated tests would not be added to the manually written tests in the code repository due to their low quality and the excessive amount. Some prior work explored improving the quality of the generated tests, for example: Helmes et al. [9] proposed to use relative LOC to guide the choosing of test generation targets; Reddy et al. [62] proposed to use reinforcement learning to guide the random input generator in property-based test generation. So far, these automated techniques are used only in addition to manually written tests. In contrast, we focus on improving developers' productivity when writing manual tests.

Another disadvantage of the automated test generation approaches is the lack of test oracles. To remedy that, prior work explored extracting test oracles from code comments, focusing on test oracles related to exceptional behaviors, null pointer checks, and boundary conditions [63]–[66]. These techniques target generating/completing test oracles, but we target completing any part of the tests, including test oracles.

Prior work also explored using deep learning models for test oracle generation without the use of comments, including ATLAS [39] and TOGA [40]. We have described both models in Section VI-B and compared TeCo with them on the task of test oracle generation, which can be considered as downstream application of our test completion task.

Tufano et al. [67] developed a code generation technique for tests based on a BART architecture pre-trained on English and code corpora. While they target to generate the entire test method as a whole, we target to complete one statement at a time, which allows the developer to observe and control the process of writing a test method.

Test recommendation. Prior work also explored improving developers' productivity in testing by test recommendation: given a method under test, suggest relevant test methods from the existing test suite using a recommendation system [68]–[71]. These techniques rely on having a set of relevant existing tests to recommend tests from, which is usually not the case when developers are starting a new project or adding tests to a project without tests. Our technique helps developers by providing completions while they are writing tests and does not have this limitation.

ML for SE. The applications of ML models on SE tasks is an active research area in recent years. One of the most studied task is code completion, which improves developers' productivity by suggesting next tokens or statements as developers are writing code [26]–[29], [35], [54], [72]–[76]. Researchers have also studied developing ML models for other SE tasks, including code summarization [31]–[33], [47], [57], [58], [77], [78], code and comment maintenance [79]–[82], bug fixing [83]–[85], etc. In this work, we propose the novel task of test completion, which brings several unique features (e.g., method under test) and necessitates reasoning about code execution. We also compared TeCo to recent work on code completion [35], [54].

Prior work explored the use of code execution data in ML for SE. Wang et al. [86] proposed to train semantic code embeddings from execution traces, which can be used to improve the performance of program repair models. Wang and Su [87] blended syntactical and semantic code embeddings and applied them in a method naming model. Nie et al. [88] developed Roosterize, a model for suggesting lemma names in verification projects which is trained using the runtime representations of lemmas. Pei et al. [89] developed a transfer learning framework called TREX that learns execution semantics from forced-execution traces to detect similar binary functions. Pi et al. [90] proposed PoEt that improves the reasoning capabilities of language models by pre-training on code execution data. Shi et al. [91] proposed to improve code generation models' outputs using a minimum Bayes risk decoding algorithm based on execution results. TeCo is the first model designed with code execution in the testing domain, specifically on the test completion task, where reasoning about the execution of the code under test is needed. Moreover, TeCo integrates execution to improve both training (using code semantics) and inference (using reranking via execution) of the model.

X. Conclusion

We introduced an idea of designing ML models for code-related tasks with code semantics inputs and reranking based on test execution outcomes. Based on this idea, we developed a concrete model, named TeCo, targeting a novel task: test completion. We evaluated TeCo on a new corpus, containing 130,934 methods and 101,965 executable methods. Our results show that TeCo significantly outperforms the state-of-the-art on code completion and oracle generation tasks, across a number of evaluation metrics. We believe that TeCo is only a starting point in the exciting area of ML for code with code semantics and execution data.

Acknowledgments

We thank Nader Al Awar, Alex Dimakis, Greg Durrett, Kush Jain, Yu Liu, Sheena Panthaplackel, August Shi, Aditya Thimmaiah, Zhiqiang Zang, Jiyang Zhang, and the anonymous reviewers for their comments and feedback. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. This work is partially supported by the US National Science Foundation under Grant Nos. CCF-1652517, CCF-2107291, IIS-2145479, and CCF-2217696.

REFERENCES

[1] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.

[2] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, *The Fuzzing Book*, 2019.

[3] Z. Zang, N. Wiatrek, M. Gligoric, and A. Shi, "Compiler testing using template java programs," in *Automated Software Engineering*, 2022, pp. 1–13.

[4] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," in *International Conference on Functional Programming*, 2000, pp. 268–279.

[5] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *International Symposium on Software Testing and Analysis*, 2002, pp. 123–133.

[6] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *International Conference on Software Engineering*, 2010, pp. 225–234.

[7] I. Kuraj, V. Kuncak, and D. Jackson, "Programming with enumerable sets of structures," in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 37–56.

[8] A. Celik, S. Pai, S. Khurshid, and M. Gligoric, "Bounded exhaustive test-input generation on GPUs," in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2017, pp. 94:1–94:25.

[9] J. Holmes, I. Ahmed, C. Brindescu, R. Gopinath, H. Zhang, and A. Groce, "Using relative lines of code to guide automated test generation for python," *Transactions on Software Engineering and Methodology*, vol. 29, no. 4, pp. 1–38, 2020.

[10] N. Al Awar, K. Jain, C. J. Rossbach, and M. Gligoric, "Programming and execution models for parallel bounded exhaustive testing," in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2021, pp. 1–28.

[11] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2010.

[12] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *International Conference on Software Engineering*, 2011, pp. 416–419.

[13] M. B. Cohen, J. Snyder, and G. Rothermel, "Testing across configurations: Implications for combinatorial testing," *Software Engineering Notes*, vol. 31, no. 6, pp. 1–9, 2006.

[14] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *International Symposium on Software Testing and Analysis*, 2017, pp. 57–67.

[15] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li, "Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs," in *Automated Software Engineering*, 2011, pp. 23–32.

[16] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *Automated Software Engineering*, 2016, pp. 625–636.

[17] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *International Conference on Software Engineering*, 2012, pp. 837–847.

[18] M. Rahman, D. Palani, and P. Rigby, "Natural software revisited," in *International Conference on Software Engineering*, 2019, pp. 37–48.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Conference on Neural Information Processing Systems*, 2017, pp. 6000–6010.

[22] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[23] 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," *Annual Meeting of the Association for Computational Linguistics*, pp. 7871–7880, 2020.

[24] 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," *Journal of Machine Learning Research*, vol. 21, pp. 1–67, 2020.

[25] S. Proksch, J. Lerch, and M. Mezini, "Intelligent code completion with Bayesian networks," *Transactions on Software Engineering and Methodology*, vol. 25, no. 1, pp. 1–31, 2015.

[26] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: AI-assisted code completion system," in *International Conference on Knowledge Discovery and Data Mining*, 2019, pp. 2727–2735.

[27] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *International Joint Conference on Artificial Intelligence*, 2018, pp. 4159–4165.

[28] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Conference on Programming Language Design and Implementation*, 2014, pp. 419–428.

[29] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *International Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[30] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[31] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Annual Meeting of the Association for Computational Linguistics*, 2016, pp. 2073–2083.

[32] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "A transformer-based approach for source code summarization," in *Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 4998–5007.

[33] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *International Conference on Software Engineering*, 2019, pp. 795–806.

[34] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021, pp. 2655–2668.

[35] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[36] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Annual Meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[37] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "CodeBLEU: A method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[38] C. Lin and F. J. Och, "Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics," in *Annual Meeting of the Association for Computational Linguistics*, 2004, pp. 605–612.

[39] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *International Conference on Software Engineering*, 2020, pp. 1398–1409.

[40] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "TOGA: A neural method for test oracle generation," in *International Conference on Software Engineering*, 2022, pp. 2130–2141.

[41] The JUnit Team, "JUnit 5," https://junit.org/junit5/.

[42] S. E. Robertson, S. Walker, and M. Beaulieu, "Experimentation as a way of life: Okapi at TREC," *Information Processing & Management*, vol. 36, no. 1, pp. 95–108, 2000.

[43] JavaParser Team, "JavaParser," https://github.com/javaparser/javaparser.

[44] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A code manipulation tool to implement adaptable systems," *Adaptable and Extensible Component Systems*, vol. 30, no. 19, 2002.

[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[46] X. V. Lin, C. Wang, L. Zettlemoyer, and M. D. Ernst, "NL2Bash: A corpus and semantic parser for natural language interface to the Linux

operating system," in *International Conference on Language Resources and Evaluation*, 2018.

[47] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Conference on Neural Information Processing Systems*, 2019, pp. 6563–6573.

[48] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Annual Meeting of the Association for Computational Linguistics*, 2016, pp. 1715–1725.

[49] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Empirical Methods in Natural Language Processing*, 2020, pp. 1536–1547.

[50] M. Nye, A. J. Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan *et al.*, "Show your work: Scratchpads for intermediate computation with language models," *arXiv preprint arXiv:2112.00114*, 2021.

[51] N. S. Keskar, B. McCann, L. R. Varshney, C. Xiong, and R. Socher, "Ctrl: A conditional transformer language model for controllable generation," *arXiv preprint arXiv:1909.05858*, 2019.

[52] The JUnit Team, "JUnit - about," https://junit.org/junit4/.

[53] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv:1909.09436*, 2020.

[54] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[55] JDK Team, "Lambda expression," https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

[56] T. Berg-Kirkpatrick, D. Burkett, and D. Klein, "An empirical investigation of statistical significance in NLP," in *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, 2012, pp. 995–1005.

[57] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, pp. 1–39, 2019.

[58] Y. Liang and K. Q. Zhu, "Automatic generation of text descriptive comments for code blocks," in *AAAI Conference on Artificial Intelligence*, 2018, pp. 5229–5236.

[59] C. Lin and F. J. Och, "ORANGE: A method for evaluating automatic evaluation metrics for machine translation," in *International Conference on Computational Linguistics*, 2004, pp. 501–507.

[60] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "PaLM: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[61] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago *et al.*, "Competition-level code generation with AlphaCode," *arXiv preprint arXiv:2203.07814*, 2022.

[62] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly generating diverse valid test inputs with reinforcement learning," in *International Conference on Software Engineering*, 2020, pp. 1410–1421.

[63] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing javadoc comments to detect comment-code inconsistencies," in *International Conference on Software Testing, Verification, and Validation*, 2012, pp. 260–269.

[64] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *International Symposium on Software Testing and Analysis*, 2016, pp. 213–224.

[65] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *International Symposium on Software Testing and Analysis*, 2018, pp. 242–253.

[66] M. Motwani and Y. Brun, "Automatically generating precise oracles from structured natural language specifications," in *International Conference on Software Engineering*, 2019, pp. 188–199.

[67] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers," *arXiv preprint arXiv:2009.05617*, 2020.

[68] W. Janjic and C. Atkinson, "Utilizing software reuse experience for automated test recommendation," in *International Workshop on Automation of Software Test*, 2013, pp. 100–106.

[69] R. Pham, Y. Stoliar, and K. Schneider, "Automatically recommending test code examples to inexperienced developers," in *International Symposium on the Foundations of Software Engineering, NIER*, 2015, pp. 890–893.

[70] R. Qian, Y. Zhao, D. Men, Y. Feng, Q. Shi, Y. Huang, and Z. Chen, "Test recommendation system based on slicing coverage filtering," in *International Symposium on Software Testing and Analysis, Tool Demonstrations*, 2020, pp. 573–576.

[71] C. Zhu, W. Sun, Q. Liu, Y. Yuan, C. Fang, and Y. Huang, "HomoTR: Online test recommendation system based on homologous code matching," in *Automated Software Engineering, Tool Demonstrations*, 2020.

[72] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, 2010.

[73] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *International Symposium on the Foundations of Software Engineering*, 2009, pp. 213–222.

[74] S. Han, D. R. Wallace, and R. C. Miller, "Code completion from abbreviated input," in *Automated Software Engineering*, 2009, pp. 332–343.

[75] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," in *International Conference on Software Engineering*, 2012, pp. 69–79.

[76] J. Lee, P. Nie, J. J. Li, and M. Gligoric, "On the naturalness of hardware descriptions," in *International Symposium on the Foundations of Software Engineering*, 2020, pp. 530–542.

[77] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, R. J. Mooney, and M. Gligoric, "Leveraging class hierarchy for code comprehension," in *Workshop on Computer Assisted Programming*, 2020.

[78] P. Nie, J. Zhang, J. J. Li, R. J. Mooney, and M. Gligoric, "Impact of evaluation methodologies on code summarization," in *Annual Meeting of the Association for Computational Linguistics*, 2022, pp. 4936–4960.

[79] S. Panthaplackel, P. Nie, M. Gligoric, J. J. Li, and R. J. Mooney, "Learning to update natural language comments based on code changes," in *Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 1853–1868.

[80] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "CoditT5: Pretraining for source code and natural language editing," in *Automated Software Engineering*, 2022, pp. 1–12.

[81] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney, "Deep just-in-time inconsistency detection between comments and source code," in *AAAI Conference on Artificial Intelligence*, 2021, pp. 427–435.

[82] S. Panthaplackel, M. Gligoric, R. J. Mooney, and J. J. Li, "Associating natural language comment and source code entities," in *AAAI Conference on Artificial Intelligence*, 2020, pp. 8592–8599.

[83] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney, "Using developer discussions to guide fixing bugs in software," in *Empirical Methods in Natural Language Processing*, 2022, pp. 2292–2301.

[84] ——, "Learning to describe solutions for bug reports based on developer discussions," in *Annual Meeting of the Association for Computational Linguistics*, 2022, pp. 2935–2952.

[85] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *Automated Software Engineering*, 2021, pp. 443–455.

[86] K. Wang, Z. Su, and R. Singh, "Dynamic neural program embeddings for program repair," in *International Conference on Learning Representations*, 2018.

[87] K. Wang and Z. Su, "Blended, precise semantic program embeddings," in *Conference on Programming Language Design and Implementation*, 2020, pp. 121–134.

[88] P. Nie, K. Palmskog, J. J. Li, and M. Gligoric, "Deep generation of Coq lemma names using elaborated terms," in *International Joint Conference on Automated Reasoning*, 2020, pp. 97–118.

[89] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Learning approximate execution semantics from traces for binary function similarity," *Transactions on Software Engineering*, pp. 1–14, 2022.

[90] X. Pi, Q. Liu, B. Chen, M. Ziyadi, Z. Lin, Q. Fu, Y. Gao, J.-G. Lou, and W. Chen, "Reasoning like program executors," in *Empirical Methods in Natural Language Processing*, 2022, pp. 761–779.

[91] F. Shi, D. Fried, M. Ghazvininejad, L. Zettlemoyer, and S. I. Wang, "Natural language to code translation with execution," in *Empirical Methods in Natural Language Processing*, 2022, pp. 3533–3546.

[92] 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," *arXiv preprint arXiv:2206.01335*, 2022.

[93] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "Adaptive test generation using a large language model," *arXiv preprint arXiv:2302.06527*, 2023.

[94] 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 *International Conference on Software Engineering*, 2023, p. to apper.

[95] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Conference on Neural Information Processing Systems*, 2022.

```
1 public GMOperation addImage(final File file) {
2   if (file == null) {
3     throw new IllegalArgumentException("file must be
          defined");
4   }
5   getCmdArgs().add(file.getPath());
6   return this;
7 }
8
9 // Here is a test for the above method. Please complete
       the next statement of the test
10 @Test public void
       addImage_ThrowsException_WhenFileIsNull() throws
       Exception {
11 exception.except(IllegalArgumentException.class);
12 // Please compete the next statement:
```

Fig. 6: The prompt to Codex for the example test completion task in Fig. 1.

APPENDIX A
COMPARISONS WITH LARGE LANGUAGE MODELS

Recent large language models for code that scale up to billions of parameters, such as Codex [30], have been shown to be promising for many code-related tasks. In parallel with our work, researchers applied large language models to generate tests [92]–[94]. In this appendix, we perform an additional experiment to evaluate the performance of large language models on test completion and compare with TECO.

The large language model we used is Codex [30], which is the state-of-the-art specialized large language model for code. Following the contemporary work on using large language models for test generation [92]–[94], we used Codex to perform test completion in the zero-shot learning setup [95], i.e., providing Codex with a prompt that contains the method under test, test method signature, and prior statements, and letting it generate the next statement. Because Codex is pre-trained to complete code, the prompt needs to be carefully designed as a code fragment to be completed. Fig. 6 illustrates the prompt format we used. We configured Codex to generate until seeing the first ';', similar to the way we used CodeGPT. Because the current generation speed of Codex is quite slow, we configured Codex to only generate the top-1 next statement using the greedy decoding algorithm. We used the `code-davinci-002` version of the Codex model. Running Codex on our evaluation set (with 30,194 statements) took 18 hours.

Table V shows the results of Codex for the test completion task with comparisons to TECO and the other baseline models; the results are organized into four parts—on the full evaluation set, runnable subset, oracle subset, and oracle-runnable subset—as explained in Section VI. TECO statistically significantly outperforms Codex on all metrics, which confirms the importance of using code semantics and code execution together with ML. Compared with the other baseline models (CodeT5/CodeGPT), Codex has better performance on some metrics (e.g., %Run on the runnable subset and oracle-runnable subset; exact-match accuracy on the oracle subset and oracle-runnable subset) but has slightly worse performance on others. Although Codex is expected to be much more powerful than CodeT5/CodeGPT due to the larger scale (billions of parameters vs. millions of parameters) and more pre-training data, we hypothesize that fine-tuning CodeT5/CodeGPT on our large test completion corpus helped with improving their performance. Codex performs better on the test oracle generation task than the test completion task, which may be because of the more prior statements context available when performing test oracle generation.

TABLE V: Results for Codex, TECO, and other baseline models on test completion. The best number for each metric is bolded. In each table, numbers marked with the same greek letter prefix are not statistically significantly different.

(a) On the evaluation set.

Model	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
Codex	12.69	N/A	34.53	29.91	58.08	56.04
CodeT5	13.57	24.11	38.33	33.88	60.81	62.40
CodeT5-noFt	0.00	0.00	0.00	3.36	1.68	0.02
CodeGPT	12.20	22.67	36.30	31.84	59.09	61.10
TECO	**17.61**	**27.20**	**42.01**	**37.61**	**63.49**	**65.23**

(b) On the runnable subset.

Model	%Compile	%Run	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
Codex	38.80	19.12	12.88	N/A	34.89	30.12	58.44	56.58
CodeT5	54.84	17.62	14.38	25.39	39.26	34.55	61.36	63.15
CodeT5-noFt	0.00	0.00	0.00	0.00	0.00	3.35	1.65	0.03
CodeGPT	53.77	15.13	12.95	24.03	37.19	32.46	59.75	61.90
TECO	**76.22**	**28.63**	**18.96**	**28.40**	**43.15**	**38.45**	**64.12**	**66.09**

(c) On the oracle subset.

Model	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
Codex	12.30	N/A	35.09	30.24	59.59	57.67
CodeT5	8.45	24.04	39.03	31.03	66.50	66.63
CodeT5-noFt	$^{\alpha}$0.00	0.00	0.00	1.18	1.86	0.01
CodeGPT	10.56	27.19	40.91	33.33	67.63	67.94
ATLAS	$^{\alpha}$0.21	0.66	21.55	13.39	54.06	50.70
TOGA	9.01	9.01	25.46	24.73	29.60	28.06
TECO	**16.44**	**27.41**	**43.09**	**35.88**	**68.05**	**68.71**

(d) On the oracle-runnable subset.

Model	%Compile	%Run	XM	Acc@10	BLEU	CodeBLEU	EditSim	ROUGE
Codex	39.46	20.73	12.15	N/A	35.05	30.12	59.79	57.86
CodeT5	44.45	16.87	8.79	24.37	40.26	32.23	67.08	67.42
CodeT5-noFt	0.00	0.00	$^{\alpha}$0.00	0.00	0.00	1.26	1.87	0.01
CodeGPT	47.39	16.13	10.41	**28.17**	41.56	33.79	67.78	68.43
ATLAS	3.62	1.45	$^{\alpha}$0.23	0.68	21.81	13.56	54.19	50.87
TOGA	25.61	9.37	9.10	9.10	26.51	25.74	31.00	29.38
TECO	**67.93**	**30.29**	**17.37**	27.39	**44.27**	**36.98**	**68.43**	**69.35**