ScenEval: A Benchmark for Scenario-Based Evaluation of Code Generation

Debalina Ghosh Paul, Hong Zhu and Ian Bayley

School of Engineering, Computing and Mathematics, Oxford Brookes University

Oxford OX33 1HX, UK. Email: hzhu@brookes.ac.uk

Abstract—In the scenario-based evaluation of machine learning models, a key problem is how to construct test datasets that represent various scenarios. The methodology proposed in this paper is to construct a benchmark and attach metadata to each test case. Then a test system can be constructed with test morphisms that filter the test cases based on metadata to form a dataset.

The paper demonstrates this methodology with large language models for code generation. A benchmark called ScenEval is constructed from problems in textbooks, an online tutorial website and Stack Overflow. Filtering by scenario is demonstrated and the test sets are used to evaluate ChatGPT for Java code generation.

Our experiments found that the performance of ChatGPT decreases with the complexity of the coding task. It is weakest for advanced topics like multi-threading, data structure algorithms and recursive methods. The Java code generated by ChatGPT tends to be much shorter than reference solution in terms of number of lines, while it is more likely to be more complex in both cyclomatic and cognitive complexity metrics, if the generated code is correct. However, the generated code is more likely to be less complex than the reference solution if the code is incorrect.

Index Terms—Machine learning; Large language models; ChatGPT; Code Generation; Benchmark; Performance evaluation; Scenario-based testing.

I. INTRODUCTION

Scenario-based testing has long been proven to be an efficient and effective testing method for traditional software and widely applied in practice. For machine learning (ML) applications, standards for developing safety critical systems, like ISO26262 [3] for road vehicles, requires the method of scenario-based testing to be applied systematically to thoroughly cover hazardous scenarios. Consequently, recent years have seen a rapid growth in research on scenario-based testing of autonomous vehicles [1], [2].

It is highly desirable, however, that scenario-based testing can be applied not only to conventional software and safety critical applications, but also to sophisticated ML models such as large language models (LLMs)[4]. A key problem is how to construct datasets that represent various scenarios efficiently and effectively. This paper addresses this problem in the context of testing and the evaluation of an LLM's capability for code generation.

This paper is organised as follows. Section II reviews existing work on the testing and evaluation of code generation, including benchmarks and the performance metrics. Section III explains how benchmark ScenEval was constructed and analyses its main characteristics. Section IV presents the datamor-phic test system for scenario-based testing with ScenEval and its implementation using the tool Morphy. Section V reports a case study with the testing and evaluation of ChatGPT. Section VI concludes the paper with a discussion of future work.

II. RELATED WORK

A. Scenario-Based Testing and Evaluation of ML

A scenario is an operational situation in a given use case of a system. For traditional software, a scenario is typically represented as a linear sequence of interactions between the user and the system. The identification and specification of scenarios is an integral part of use case driven software engineering [5]. Test data can then be derived from the sequence of human-computer interactions through instantiation. In contrast, for ML applications, it is often a category of input queries given to the ML model that represents the same operation situation. Therefore, traditional scenario-based testing techniques cannot be applied straightforwardly.

In order to address this problem, Zhu *et al*. [6] proposed a process model for identifying scenarios in the operation of ML applications, and defined a set of test adequacy criteria to cover combinations of scenarios. In [4], Zhu *et al*. advanced an automated technique for generating test data. It employs data augmentation operators known as datamorphisms to transform test data of a *seed scenario* to a *mutant scenario*.

This technique was then applied to deep neural networks (DNN) for computer vision, specifically the perception system of an autonomous racing car. By evaluating the system on various scenarios, the worst performing scenarios were identified and the DNN re-trained with additional data for those scenarios and its performance improved.

While these experiments demonstrated the effectiveness and efficiency of the approach, its applicability requires the existence of data for *seed scenarios* and datamorphisms to transform them to mutant scenarios. In the case of autonomous vehicles, these are difficult to obtain. Much research efforts have been spent on simulation of different traffic and road conditions [1], [2]. However, as far as we know, there is little work on constructing suitable benchmarks for scenario-based testing in other ML application domains.

B. Benchmarks for Code Generation

This paper concerns benchmarks for the evaluation of ML models as code generation tools. Each element of the dataset

contains a natural language input that specifies the programming task. The ML model is expected to generate a piece of program code that meets the specification.

We have identified 11 different such benchmarks in the literature. They differ from each other in the way that the data are procured, the contents contained in each element, the type of code to be generated and the target language. Their key features are summarised in Table I.

The benchmarks differ in the contents of each element of the dataset. Natural language descriptions are always present, usually given as docstrings. In addition, function signatures and unit test cases may be provided, both of which are used for test automation to check the correctness of generated solutions. In some cases, there may also be reference solutions. Table I also gives the contents provided by each benchmark; the column #Tests gives the average number of test cases per task if any are provided. In some cases, programming tasks in the dataset are classified into subsets of different difficulty levels. The column Difficulty levels shows the number of difficulty levels that the test cases were classified into.

C. Evaluation of Code Generation Capability

In general, the evaluation of a ML model involves activities at two levels: the individual test case level, where the ML model's output on each test case is evaluated, and the benchmark dataset level, where the overall performance is calculated from the assessments of the individual test cases.

There are two different approaches for evaluating the quality of code. The first approach is to measure its syntactic closeness to a reference solution, which can be done with the BLEU metric. However, Kulal et al. [22] found that BLEU fails to reflect functional correctness and Hendrycks et al. [13] showed that it is even inversely correlated with it. In 2020, Ren et al. [23] introduced, as an alternative to BLEU, the measure CodeBLEU, which compares abstract syntax trees and data flow graphs instead of program text. Lai et al. used a much-relaxed form of similarity metric called surface-form constraints, which identifies keywords and the presence or absence of certain APIs [12].

The second approach, proposed by Kulal et al., is to measure functional correctness instead and regard generated code as correct if it passes all test cases; Hendrycks et al. [13] , in contrast, measured the *percentage* of test cases passed.

Kulal et al. measured the overall performance with the percentage of coding tasks for which the LLM produces at least one correct solution when asked to produce 100 solutions. This is later generalised to $k > 0$ solutions for each task and the definition of the $pass@k$ metric, which is the probability of generating at least one solution in k successfully. Chen et al. [14] found that the $pass@k$ metrics produce a high variance, however, so they counted the number c of successful solutions in k and used c and k to make an unbiased estimation of the $pass@k$ metric. That has been used by most of the benchmarks reported above.

Miah and Zhu [20], in contrast, considered the use of a LLM model to be an interactive process in which the user makes a number of attempts by entering and revising the input to the LLM until a successful solution is generated, or gives up after a maximal number k of allowed attempts. This is notably different from k different solutions of one attempt. They proposed a new metric $\#attempt_k$, which is the average number of attempts over the benchmark.

Table II provides detailed information regarding the benchmarks used in the evaluation, the metrics used and the main results.

The only quality attribute considered above is correctness except that a work by Miah and Zhu [20] assessed structuredness, conciseness, completeness, and logical clarity, as well as attributes on the textual explanation of the code.

In summary, existing benchmarks for code generation fall short in their support for scenario-based testing since none of them link test cases to scenarios that might be encountered during code generation. The solution proposed by Miah and Zhu [20] was to include metadata with each test case to represent the scenario it tests.

This paper further develops that metadata approach by constructing a large scale benchmark ScenEval with 12864 Java programming tasks, of many different kinds and from different sources, all tagged with scenario information as metadata in JSON format. With the support of the automated testing tool Morphy [7], the concept of test morphisms makes it easy to form datasets with test cases that all belong to one scenario and, by changing and combining datasets, to study the performance of a LLM across several different scenarios.

III. SCENEVAL BENCHMARK

We now describe the structure of each task within ScenEval and then describe the scenarios it handles. All tasks are labelled with scenario information to improve upon existing work and for the same reason, a variety of sources have been used.

A. Structure of Data

In ScenEval, each test case is a coding task with the following metadata represented as a JSON value whose structure is given in Figure 1.

- Task Id: an universal unique identifier of coding task;
- Title: the title of the coding task;
- Source: a list of sources, which can be more than one if the task occurs in more than one sources.
- Topics: a list of topics covered by the coding task.
- Programming language: The programming language in which the code is to be generated.
- Version: The version number of the task, to support the evolution of the dataset.
- Description: A description of the coding task, which can be a sequence of text or a code snippet (such as a signature or a skeleton), or a fully qualified file name for image data; see Figure 2.
- Reference Solutions: A list of reference solutions.

Currently, we distinguish three types of sources for the coding tasks: textbook, real-world questions, and synthetic

TABLE I: Main Features of Existing Benchmarks

Benchmark	Source	Level	#Tasks	Language	Signature	#Tests	Solution	Difficulty Levels
APPS[13]	Coding challenge	Program	10,000	Python	-	+	+	3
HumanEval[14]	Domain Experts	Function	164	Python	+	7.7	-	-
MBPP[15]	Crowd-sourcing	Function	974	Python	-	3	-	-
MathQA-Python[15]	MathQA	Function	23,914	Python	-	3	-	-
ClassEval[19]	Repository, HumanEval, MBPP	Class	100	Python	+	33.1	+	-
CoderEval[10]	Github	Function	230	Python	+	+	-	6
		Method	230	Java				
Multipl-E[16]	HumanEval, MBPP	Function	1138	Various	+	3 to 7	-	-
DS-1000 [12]	Stack Overflow	Statement	1000	Python	+	1.6	+	-
HumanEval+[17]	HumanEval	Function	164	Python	+	774.8	-	-
CONCODE [11]	Github	Method	2000	Java	+	-	-	-
R-benchmark[20]	Text Books	Program	351	R	-	-	+	3

TABLE II: Uses of Benchmark In Evaluations

Benchmark	Correctness	Perf. Metrics	ML Model	Result
APPS	pass all tests, %passed tests	%pass@100, Avg %passed	GPT-2 GPT-Neo GPT-3	0.68, 7.96 1.12, 10.15 0.06, 0.55
HumanEval	pass all tests	%pass@100	GPT-Neo Codex	21.37 72.31
MBPP	pass all tests	%pass@1	Decoder Transformer Lang. Model	79.0 82.8 83.8
MathQA -Python	pass all tests	%pass@1	Decoder Transformer Lang. Model	74.7 79.5 81.2
ClassEval	pass all tests	%pass@5	GPT-4 GPT-3.5 CodeGen	42.0 36.0 13.0
CoderEval (Python)	pass all tests	%pass@10	CodeGen ChatGPT	23.48 30.00
CoderEval (Java)	pass all tests	%pass@10	CodeGen ChatGPT	33.48 46.09
Multipl-E (HumanEval)	pass all tests	%pass@1	Codex CodeGen	≈ 36 ≈ 9
Multipl-E (MBPP)	pass all tests	%pass@1	Codex CodeGen	≈ 40 ≈ 14
DS-1000	pass all tests	%pass@1	Codex-002 CodeGen	41.25 8.4
HumanEval+	pass all tests	%pass@100	CodeGen ChatGPT GPT-Neo	≈ 64.0 89.8 16.8
ConCode	BLEU	Avg BLEU	Retrieval Seq2Seq Seq2Prod	20.27 23.51 21.29
R-benchmark	Satisfactory	Avg #attempt$_k$	ChatGPT	1.6

Fig. 1: Structure of JSON representation of Task

Fig. 2: Structure of JSON representation of Description and Type
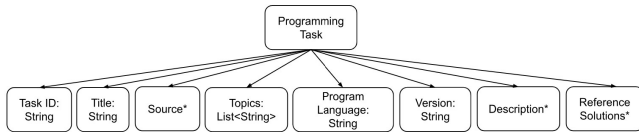

Fig. 3: Structure of JSON Representation of Various Types of Sources

data. The metadata structure for each source type is given in Figure 3. More types of sources can be easily added due to the extensibility of JSON.

We allow multiple reference solutions to be provided for each coding task. As shown in Figure 4, each reference solution is also associated with metadata for its source and complexity. Three metrics are used for the latter: cyclomatic complexity, cognitive complexity and the number of lines.
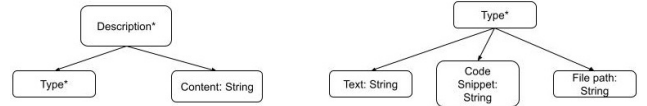
Figure 5 shows an example of the test cases in ScenEval.

B. Data Procurement and Extraction

The tasks of ScenEval are extracted from three types of sources:

- *Textbook*: Exercises and solutions in four textbooks on Java programming [24], [25], [26], [27].
- *Online learning website*: Exercise questions and solutions on Java programming at the website of W3Resources[1].
- *Online forum*: Questions and answers about Java programming posted on Stack Overflow[2].

Data in the textbooks are extracted manually and metadata are also assigned manually. Exercise questions that do not require code to be written are excluded. There are a total of 1306 tasks from 4 textbooks; see Table III for the number of tasks extracted from each source.

For the online sources, data together with the metadata are extracted automatically by running script codes. Tasks from W3Resource have questions that are well presented as exercises for students who are learning programming and the solutions are tested and reliable. Therefore, in the sequel, they are also categorised as *textbook questions*.

[1]URL: https://www.w3resource.com/java-exercises/
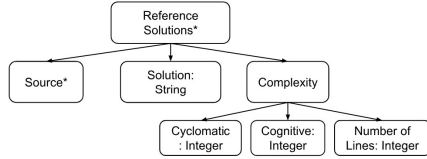[2]URL: https://stackoverflow.com/

Fig. 4: Structure of JSON Representation of Reference Solutions

```
{"Source": {
    "sourceType": "Text Book",
    "bookName": "Introduction to Java Programming",
    "authors": "Y. Daniel Liang",
    "chapterName": "Abstract Classes and Interfaces"},
"problemTitle": "The ComparableCircle class",
"includedTopics": ["Abstract Classes", "Interfaces"],
"problemID": "2e51f976-d621-4e5f-af52-bb061060727c",
"version": "v1.0",
"programLanguage": "java",
"parts": [ { "type": "text",
    "content": "Define a class named ComparableCircle that extends Circle and implements Comparable. For the
    Circle class, tasks include defining instance variables to represent properties such as radius or diameter,
    implementing methods to calculate properties like area and circumference, providing constructors for object
    initialization, and possibly including methods for setting or getting radius/diameter values. For the
    Comparable interface, tasks involve defining the compareTo method to establish the natural ordering of
    objects, implementing it to compare the areas of two ComparableCircle objects, ensuring it returns a
    negative integer if the current object is smaller, zero if they are equal, and a positive integer if the current
    object is larger, and overriding the equals method for consistency with compareTo. Write a test class to
    determine which of two ComparableCircle objects has a larger area.}],
"testCases": [ ],
"GivenSolutions": [
    {"SourceType": "Text Book",
    "SourceName": "Introduction to Java Programming",
    "solution": [
        { "partsOfSolution": "C:\\ScenEval\\ChatGPT_generated_programs\\w3resourcesAbstractClass
            \\2e51f976-d621-4e5f-af52-bb061060727c\\GivenSolutionOnePart1.java",
        "LOC": "38",
        "CognitiveComplexity": "3",
        "CyclomaticComplexity": "4" } ] },
    {"SourceType": "Web Source",
    "SourceName": "w3school",
    "solution": [
        {"partsOfSolution": "C:\\ScenEval\\ChatGPT_generated_programs\\w3resourcesAbstractClass
            \\2e51f976-d621-4e5f-af52-bb061060727c\\GivenSolutionTwoPart1.java",
        "LOC": "30",
        "cognitiveComplexity": "2",
        "CyclomaticComplexity": "3"}]}}}
```

Fig. 5: Example of a Test Case

The questions from Stack Overflow are also extracted automatically, but manually inspected to remove those not on code generation, but apart from that, neither the questions nor their solutions were edited, since we believe it is best to keep them both in their natural form for testing LLMs, even though the solutions are untested and there is little quality control other than user votes and feedback. These tasks are referred to as *real-world questions* in the sequel.

Figure 6 shows the distribution of coding tasks across different topics. It is worth noting that the real-world questions were posted to Stack Overflow over a period of time since 2008. Figure 7 shows the distribution of these tasks across the time.

Figure 8 shows the distributions of tasks by complexity according to cyclomatic complexity, cognitive complexity and lines of code (LOC), respectively.

IV. DATAMORPHIC TEST SYSTEM

The methodology of datamorphic testing regards software testing as a problem of systems engineering, and aims to develop and apply a test system, in which testing is performed, test resources are managed and testing processes are automated.

TABLE III: Number of Tasks from Various Sources

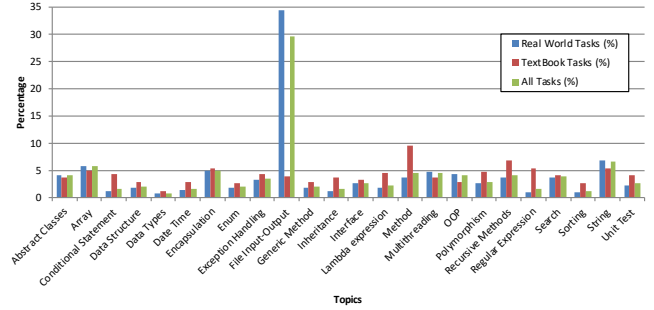| Source	#Tasks
W3Resources	1058
Stack Overflow	10500
Textbooks	1306
Introduction to Java Programming by Y. Daniel Liang	791
Absolute Java by Walter Savitch	217
Java: A Beginner's Guide by Herbert Schildt	230
Programming and Problem Solving with Java by Nell Dale et al.	68
Total	**12864**

Fig. 6: Distribution of Coding Tasks by Topic

Moreover, datamorphic testing constructs test systems by classifying the artefacts of testing into two types: test entities (such as test data, objects, the software under testing, test results, etc.) and test morphisms, which are operations on or transformers of test entities. Here, benchmarks are test entities and so are the various subsets of a benchmark that represent various scenarios.

The tool Morphy has been developed to support datamorphic testing [7]. The tester can define test entities in Java as classes and implement test morphisms as Java methods. In addition to general test actions, Morphy recognises the following types of test morphisms:

- *Seed maker*: which generates test cases from other types of entities;
- *Datamorphism*: which transforms test cases;
- *Metamorphism*: which checks the correctness of test cases and output a Boolean value;
- *Test set filter*: which add or remove test cases from a test set;
- *Test set metric*: which maps a test set to a real value, such as its test adequacy;
- *Test case filter*: which maps a test case to a Boolean value to decide whether to keep it in the test set;
- *Test case metric*: which maps test cases to a real value, such as its complexity.
- *Analyser*: which analyses the test set and produces a test report.
- *Executer*: which invokes the program under test with input data from the test case and receives the output from the program.

In addition to providing facilities to manage test entities, Morphy supports test automation at three levels, each of which
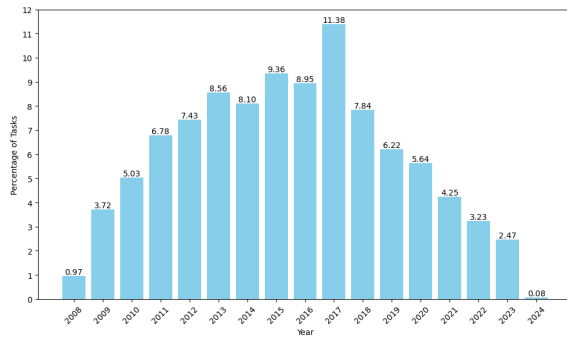
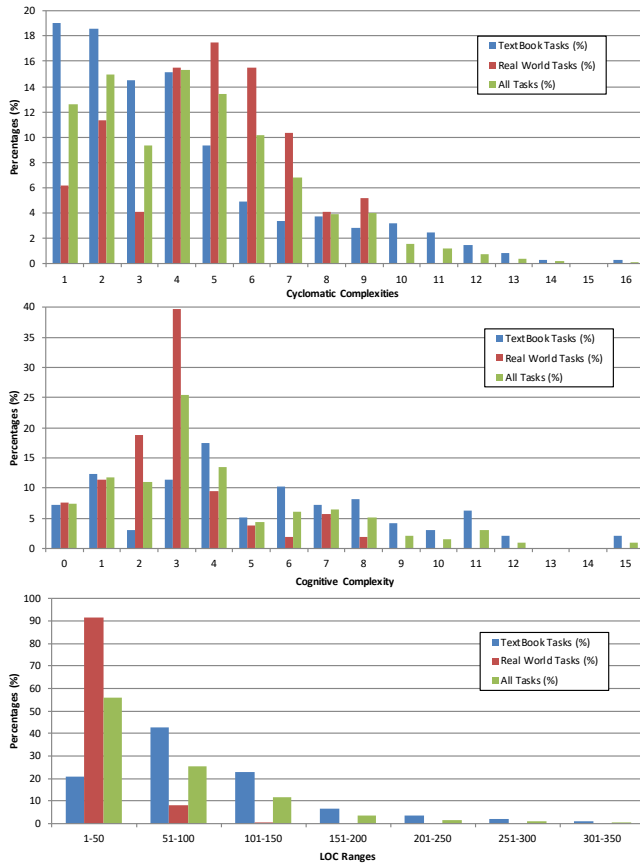Fig. 7: Distribution of Real-World Questions by Year

Fig. 8: Distribution of Coding Tasks

can be invoked with a button click:

- *action*, the level of a single test activity;
- *strategy*, a composition of test morphisms, which can be defined as an algorithm that contains test morphisms as parameters;
- *process*, a sequence of invocations of actions and strategies that can be recorded into an editable test script.

To support scenario-based testing in particular, we defined and implemented four different sorts of test morphisms: test set filters, analysers, seed makers and test executors. We will now examine each of these.

A. Test Set Filters.

Four test set filters have been implemented to select test tasks in a dataset according to the source, topics, complexity and years.

- *filterBySource*, which filters test cases according to the source type supplied as input; if that source type is 'text book', the user can select a specific textbook
- *filterByTopic*, which filters test cases according to a set of topics supplied as input
- *filterByComplexity*, which filters test case according to complexity metric and a range within that metric
- *filterByYears*, which filters test cases according to a range of years, specified with a start year and end year.

These filters can be combined. For example, you can create a dataset with topics on threads from real-world questions after the year 2010, by applying filterByTopic then filterBySource then filterByYear. Datasets can be saved and then loaded to be combined with other datasets.

B. Test Data Analysers.

There are two types of analysers, those that analyse the data distributions and those that analyse each test case on various quality attributes.

1) Analysers of data distributions.: Three analysers have been implemented to calculate and display the data distribution in the dataset according to the topic, year and complexity respectively:

- *TopicBasedDistribution*, example outputs of which can be seen in Figure 6.
- *YearBasedDistribution*, as seen in Figure 7.
- *ComplexityBasedDistribution*, as seen in Figure 8.

2) Analysers of test cases features.: Seven analysers have been implemented to analyse various aspects of the quality of each test task in the dataset.

- *isCodeCompilable*, which takes the generated code snippet, compiles it and returns a message saying whether the compilation was successful.
- *isCodeExecutable*, which takes the object code and executes it to show the output.
- *analyseComplexity*, which calculates the cyclomatic complexity and cognitive complexity of the solution and saves it in the metadata along with the number of lines. The function is implemented by invoking the PMD code analyser[3] and extracting the output produced.
- *generateTestCode*, which generates two JUnit test classes, one for the reference solution and the other for the generated solution. It is implemented by invoking Evosuite [8], [9].
- *purifyReferenceTestCode*, which runs all unit tests on the reference solution, reports the correctness of each one and removes the failing test cases
- *purifySolutionTestCode*, as for *purifyReferenceTestCode*, but applied to the solution test code.

---

[3]URL: https://pmd.github.io/

- *runTestCodes*, which runs the test cases on both the reference solutions and generated solutions, reports on the correctness of the latter, and then measures test adequacy according to various code coverage metrics by invoking the functions provided by Evosuite [8], [9]. Figure 9 shows the average test coverage of the reference solutions on the whole ScenEval benchmark as an example of the output from this analyser.
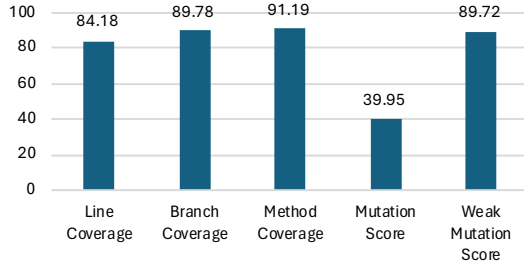

Fig. 9: Average Test Coverages

C. Seed Makers

Three seed makers have been implemented.
- *ManualTaskEntry*, for manually entering test tasks, as used for those in textbooks.
- *ExtractStackOverflow*, to extract tasks from Stack Overflow.
- *ExtractW3Resource*, to extract data from W3Resource.

Once the data has been extracted and stored in Morphy, it can be inspected and those tasks not suitable for code generation can be removed. Data can be edited, but it has not been for the current version of the benchmark.

D. Test Executer

Only one test executer has been implemented: *ChatGPTTestExecuter*, which invokes ChatGPT through its API, sends requests taken from the test task descriptions and extracts data from the responses. Other LLMs can be tested by writing their own executers which will have different APIs, URL and data formats for request and response messages.

V. EVALUATION OF CHATGPT

This section demonstrates the use of ScenEval in forming various subsets with the support of Morphy so that a scenario-based evaluation can be carried out on ChatGPT.

A. Design of The Experiments

1) Test Case Generation: Test cases are generated by applying the *generateTestCode* test morphism. From the reference solution, it produces a JUnit test class with test cases that come with expected outputs that are checked using assertions. The test cases are not always correct, however, so the ones that failed on the reference solution are removed, by invoking the test morphism *purifyReferenceTestCode*.

It is worth noting that test case generation from the reference solution only uses the information contained in the reference code. It can be effective to check the correctness of a generated solution on the domain that the reference solution defines. Defects of a generated solution on this domain are called *omission errors* in software testing literature. However, a generated solution may also provide functions beyond this required domain, such as those containing malicious code. The test cases generated from the reference solution cannot detect such defects, which are called *commission errors*. In other words, a test code generated from the reference solution is effective to detect omission errors, but not commission errors.

To address this problem, we also generate the second test code, but from the generated solution. Similarly, the test morphism *purifySolutionTestCode* is applied to the generated code to remove the incorrect test cases. This test code when applied to the reference solution can detect commission errors.

In the sequel, the purified the test code generated from the reference solution is called the γ test, while the purified test code generated from the generated solution is called κ test.

2) Test Execution And Result Analysis: Once the γ and κ test codes are ready for a test task, both γ and κ test codes are applied to both reference solutions and generated solutions. This is by invoking the test morphism *runTestCodes*, which executes the test codes and reports the test results together with the test adequacy measured in various coverage metrics.

3) Correctness Criteria and Performance Metrics: We use two correctness and performance metrics to measure the performance of ChatGPT's code generation capability. The first is passing all test cases as the correctness criterion and percentage of $pass@1$ as the overall performance metric. Here, a generated solution is regarded as correct by passing all test cases, if both the reference and the generated solutions pass all test cases in the γ and κ tests.

The other performance metric is the average pass rate over all tasks, where pass rate for a task is calculated from the failure rate by the formula $1 - failurerate$. The failure rate for a task is the proportion of test cases on which either reference solution fails or the generated solution fails over the total number of test cases in γ and κ tests.

B. Correctness Analysis

Our main research goal is to gain insight on how ChatGPT performs on textbooks and real-world questions. Applying the scenario-based evaluation techniques, we created two test datasets; one with tasks from the textbook questions and the other with tasks from the real-world questions by applying the test morphism *filterBySources*. Then, ChatGPT is tested on these two datasets. On textbook tasks, the percentage of $pass@1$ is 75.64%, i.e. about three-quarters of the tasks correctly passed all test cases, with an overall average pass rate of 82.4%. In contrast, the percentage of $pass@1$ for real-world tasks is 67.07%, and the overall average pass rate was 74.34%.

In addition to the overall performances on these two scenarios, we further analysed ChatGPT's performance on tasks of various topics and complexity. Again, by applying the principles of scenario-based evaluation, we split the two test

datasets according to topics and complexities using the test morphisms *filterByTopics* and *filterByComplexity* and obtained two sets of sub-datasets. The performances of ChatGPT is evaluated on the datasets. The results are shown in Figure 10 and 11 respectively.
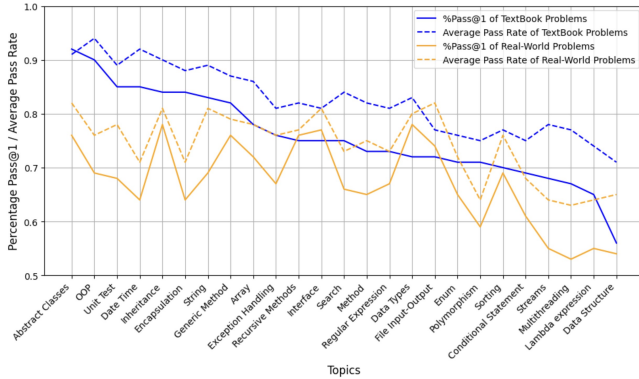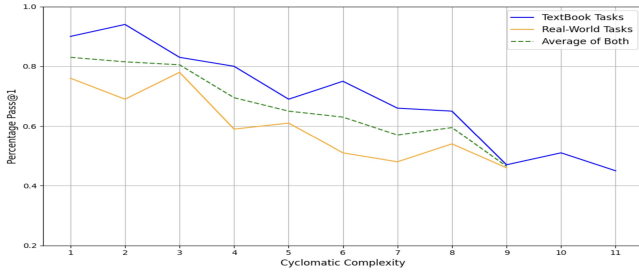

Fig. 10: Variation of Performance Over Topics

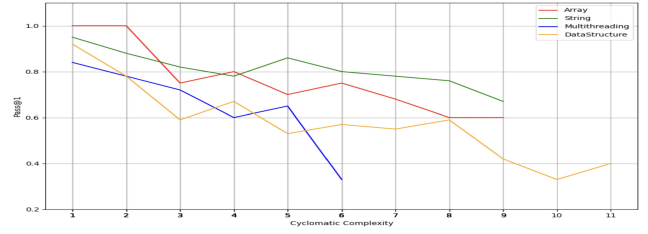Fig. 11: Variation of Performance over Cyclomatic Complexity

As shown clearly in Figure 11, ChatGPT's performance decreases as the cyclomatic complexity of the task increases. It might be just a coincidence that some topics are more complicated than others, however, so we pick the worst-performed topics and see whether we observe the same phenomenon.

From the results of the evaluation shown in Figure 10, we can identify that ChatGPT performed worst on the topics of streams, multi-threading, lambda expressions and data structures.
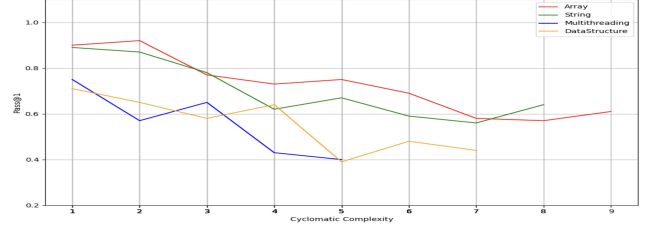
To gain further insight on ChatGPT's performance on these topics, we split each dataset of these topics into a set of sub-datasets according to the cyclomatic complexity. The evaluation on these sub-datasets revealed that the decline of performance with complexity is uniform for all topics; see Figure 12. Therefore, it is not a coincidence.

C. Complexity Analysis

To analyse the quality of the generated code, we compare the complexity of the generated solutions with that of the reference solutions. We construct a random subset of the benchmark, and submit the tasks of the dataset to ChatGPT. The results from ChatGPT are analysed by invoking the test morphism *analyseComplexity*. Figure 13 shows the distributions of cyclomatic and cognitive complexities of the reference
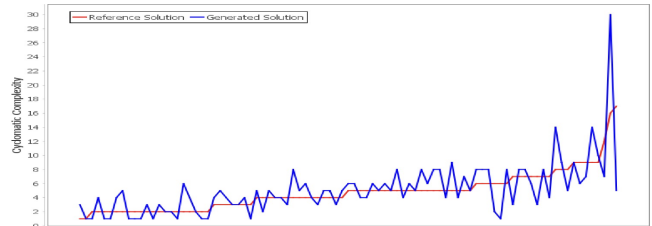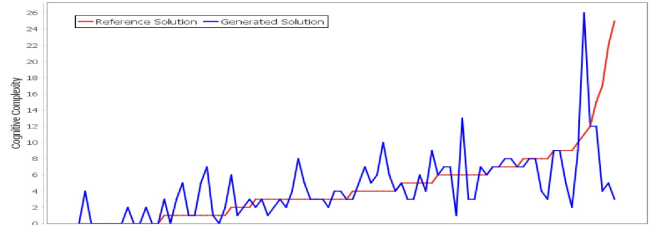

(a) Performance on Textbook Questions.

(b) Performances on Real-World Questions.

Fig. 12: Variation of Performance Over Complexity on Hard Topics

solutions and the generated codes, where red lines are the complexities of reference solutions while the blue lines are the complexities of generated codes.

(a) Comparison on Cyclomatic Complexity

(b) Comparison on Cognitive Complexity

Fig. 13: Complexity of Reference Vs Generated Solutions

Statistical analysis of the complexity data is given in Table IV, where columns AvgRS and AvgGS are the average complexities of the reference and generated solutions, respectively. The columns %Above, %Equal and %Below are the percentages of the tasks that the generated solutions have a higher, equal and lower complexity than the reference solutions, respectively. The column Avgδ gives the average

TABLE IV: Complexity of Generated vs Reference Solutions

Complexity	AvgRS	AvgGS	%Above	%Equal	%Below	Avgδ
Cyclomatic	5.36	5.52	48.45	18.56	32.99	2.31
Cognitive	5.21	5.01	41.24	25.77	32.99	2.62
LOC	33.52	24.16	33.03	7.24	58.82	14.47
CLOC	39.18	24.84	24.89	4.52	69.68	18.29
CL	5.72	0.84	14.93	27.15	57.014	5.59

TABLE V: Complexities of Generated and Reference Solutions for Correct and Incorrect Tasks

Complexity	Correct Subset						Incorrect Subset					
	AvgRS	AvgGS	%Above	%Equal	%Below	Avgδ	AvgRS	AvgGS	%Above	%Equal	%Below	Avgδ
Cyclomatic	5.25	5.69	60.66	13.11	26.23	1.92	6.00	4.81	27.78	27.78	44.44	2.97
Cognitive	5.26	5.36	39.34	29.51	31.15	2.23	5.14	4.69	36.11	19.44	44.44	3.67
LOC	31.87	23.97	34.23	9.01	54.95	12.45	35.15	24.34	31.82	5.45	62.73	16.48
CLOC	40.19	24.52	15.32	3.60	79.28	17.16	38.17	25.15	34.55	5.45	60.00	19.41
CL	8.43	0.74	3.60	6.31	88.29	7.80	3.04	0.93	26.36	25.45	48.18	3.40

absolute differences between the complexities of the generated and reference solutions. The rows of Cyclomatic and Cognitive are the data of cyclomatic and cognitive complexities. The row LOC gives the data about the numbers of lines of the code after comments are removed, CLOC is about the number of lines with comments, and CL is about the number of lines that contain comments.

From the data given in Table IV, the generated solutions are often more complex than the reference solutions. The average cyclomatic complexity of the generated solutions was 5.52, while the average of reference solutions was only 5.36. On 48.45% of the tasks the generated solutions have higher cyclomatic complexity than reference solutions, while only 32.99% tasks have lower cyclomatic complexity, and 18.56% tasks were of equal complexity. For cognitive complexity, the generated solutions have a slightly lower average cognitive complexity than that of the reference solutions. However, there are more cases that generated solutions have a higher cognitive complexity than reference solutions.

Our experiment data also shows that the generated codes are much shorter than the reference solutions in terms of the number of lines of code. On average the generated code is more than 14 lines shorter. This contradicts the observation made by Miah and Zhu on the R program code generated by ChatGPT [20]. Moreover, the generated code contains little comments. On average there is only less than 1 line in the generated code.

To further investigate the complexity of ChatGPT generated code, we split the test dataset into two subsets according to the functional correctness: one contains the tasks that the generated code passes all test cases, and the other for those that the generated code fails on tests. The statistical data is shown in Table V.

The data in Table V show that the correctly generated solutions are likely to be more complex than reference solutions in both cyclomatic and cognitive complexity metrics, and longer as well. However, for incorrectly generated codes, it is more likely to be less complex than the reference solution and more likely to be shorter.

D. Discussion

From the experiments, we make the following observations.

First, due to the metadata associated with the task in the ScenEval benchmark and the support of datamorphic testing tool Morphy, scenario-based testing and evaluation can be conducted efficiently and effectively. Benchmarks with metadata and a carefully developed test system with test morphisms for dataset filtering, test result analysis and test data distribution analysis form a powerful scenario-based test and evaluation environment, in which experiments can be conducted efficiently and effectively.

Second, using scenario-based evaluation, one can gain insight into an LLM model effectively. For example, we can identify the task topics on which ChatGPT performed poorly. These topics are the areas that ChatGPT should improve. It has been observed already that when the complexity of the tasks increases, the LLM's performance decreases. However, existing works are based on an informal judgement of the difficulty of tasks [20]. In our experiments, the complexity of tasks are measured by cyclomatic complexity and performances are evaluated on subsets of different complexities. It is further confirmed that the decrease in performance on complexity is not a coincidence because the complexity of tasks in certain topics are more complicated than other topics.

Finally, the Morphy testing tool makes the test system easy to manage and operate and flexible to extend and evolve. Various software engineering tools can be easily integrated together via implementation of invocations of existing tools and code to extract data saved by such tools. Our test system has integrated PMD and EvoSuite tools.

VI. Conclusion

We proposed a new approach to structure benchmark datasets with metadata to represent the usage scenarios of each element of the benchmark and to develop a test system for using metadata to support scenario-based testing and evaluation of LLMs. We have demonstrated how metadata makes it possible to formulate scenarios efficiently and how scenarios contribute to a thorough analysis of LLM performance.

For future work, we are further developing the test system with more test morphisms for analysis of the quality of program code. The work reported in this paper has only considered correctness and complexity. Many other aspects of code qualities can and should be evaluated, and this is already in progress.

We are also working on the testing and evaluation of many other LLMs for code generation. This can be implemented fairly easily by writing a test morphism for executing each LLM to invoke the model with queries extracted from the description of the task and to receive the responses from the LLM. Comparisons between many LLMs could be time consuming and labour intensive when the process involves manual processing of data. We are working on how to automate the whole process.

REFERENCES

[1] S. Riedmaier, T. Ponn, D. Ludwig, B. Schick and F. Diermeyer, "Survey on scenario-based safety assessment of automated vehicles", IEEE access, vol. 8, pp. 87456-87477, 2020.

[2] W. Ding, C. Xu, M. Arief, H. Lin, B. Li and D. Zhao, "A survey on safety-critical driving scenario generation-A methodological perspective", IEEE Transactions on Intelligent Transportation Systems, 2023.

[3] The British Standard Institution, Road Vehicles - Functional Safety, Part 6 Product Development at The Software Level, BS ISO 26262-6:2018.

[4] H. Zhu, T. M. T. Tran, A. Benjumea and A. Bradley, "A Scenario-Based Functional Testing Approach to Improving DNN Performance", 2023 IEEE International Conference on Service-Oriented System Engineering (SOSE 2023), July 17 - 20, 2023, Athens, Greece.

[5] I. Jacobson, Object Oriented Software Engineering: A Use Case Driven Approach, Addison Wesley, 1992.

[6] H. Zhu, D. Liu, I. Bayley, R. Harrison and F. Cuzzolin, "Datamorphic Testing: A Method for Testing Intelligent Applications," 2019 IEEE International Conference On Artificial Intelligence Testing (AITest 2019), Newark, CA, USA, 2019, pp. 149-156, doi: 10.1109/AITest.2019.00018.

[7] H. Zhu, I. Bayley, D. Liu, X. Zheng, "Morphy: A Datamorphic Software Test Automation Tool", https://doi.org/10.48550/arXiv.1912.09881.

[8] G. Fraser, A. Arcuri, "A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite", ACM Transactions on Software Engineering and Methodology, 24(2), Article 8, Dec. 2014.

[9] G. Fraser, A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software", In Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Sept. 2011, pp416–419.

[10] H. Yu, et al., "CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models", In Proc. of the IEEE/ACM 46th International Conference on Software Engineering (ICSE'24), Article 37. https://doi.org/10.1145/3597503.3623316.

[11] S. Iyer, I. Konstasy, A. Cheung and L. Zettlemoyer, "Mapping Language to Code in Programmatic Context", In Proc. of the 2018 Conference on Empirical Methods in Natural Language Processing, pp 1643–1652.

[12] Y. Lai, et al., "DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation", In Proc. of the 40 th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

[13] D. Hendrycks, et al., Measuring Coding Challenge Competence With APPS. In Proc. of 35th Conference on Neural Information Processing Systems - Datasets and Benchmarks Track, 2021.

[14] M. Chen, et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.

[15] J. Austin, et al., "Program synthesis with large language models," arXiv preprint arXiv:2108.07732, 2021.

[16] F. Cassano, et al., "MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation", IEEE Transactions on Software Engineering, 49(7), July 2023.

[17] J. Liu, C. S. Xia, Y. Wang, L. Zhang, "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation", https://doi.org/10.48550/arXiv.2107.03374.

[18] A. Amini, e tal., MathQA: Towards Interpretable Math Word Problem Solving with Operation-Based Formalisms. In Proc. of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp2357–2367, Minneapolis, Minnesota. Association for Computational Linguistics, 2019.

[19] X. Du, et al., "ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation", https://doi.org/10.48550/arXiv.2308.01861.

[20] T. Miah and H. Zhu, "User-Centric Evaluation of ChatGPT Capability of Generating R Program Code", https://doi.org/10.48550/arXiv.2402.03130.

[21] K. Papineni, S. Roukos, T. Ward, and W-J. Zhu, "BLEU: a Method for Automatic Evaluation of Machine Translation", Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL), Philadelphia, July 2002, pp311-318.

[22] S. Kulal, et al. 2019. Spoc: Search-based pseudocode to code, Advances in Neural Information Processing Systems, Volume 32. Curran Associates, Inc., 2019.

[23] S. Ren, et al. 2020. Code-bleu: a method for automatic evaluation of code synthesis. CoRR, abs/2009.10297, 2020. URL https://arxiv.org/abs/2009.10297.

[24] Y. D. Liang, Introduction to Java Programming, 10th Ed., Pearson, 2014.

[25] W. Savitch, Absolute Java, 6th Edition, Pearson, 2015.

[26] H. Schildt, Java: A Beginner's Guide, McGraw-Hill, 2022.

[27] N. Dale, C. Weems, and M. Headington, Programming and Problem Solving with Java, Jones and Bartlett Publishers, 2003.

[28] S. Kabir, D.N. Udo-Imeh, B. Kou, and T. Zhang, Who Answers It Better? An In-Depth Analysis of ChatGPT and Stack Overflow Answers to Software Engineering Questions, 2023. arXiv preprint arXiv:2308.02312.