Test Wars: A Comparative Study of SBST, Symbolic Execution, and LLM-Based Approaches to Unit Test Generation

Abdullin, A.M.; Derakhshanfar, Pouria; Panichella, Annibale

DOI
10.1109/ICST62969.2025.10989033

Publication date
2025

Document Version
Final published version

Published in
Proceedings of the 2025 IEEE Conference on Software Testing, Verification and Validation (ICST)

Test Wars: A Comparative Study of SBST, Symbolic Execution, and LLM-Based Approaches to Unit Test Generation

Azat Abdullin
JetBrains Research, TU Delft
Amsterdam, The Netherlands
azat.abdullin@jetbrains.com

Pouria Derakhshanfar
JetBrains Research
Amsterdam, The Netherlands
pouria.derakhshanfar@jetbrains.com

Annibale Panichella
TU Delft
Delft, The Netherlands
a.panichella@tudelft.nl

2025 IEEE Conference on Software Testing, Verification and Validation (ICST) | 979-8-3315-0814-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICST62969.2025.10989033

***Abstract*—Generating tests automatically is a key and ongoing area of focus in software engineering research. The emergence of Large Language Models (LLMs) has opened up new opportunities, given their ability to perform a wide spectrum of tasks. However, the effectiveness of LLM-based approaches compared to traditional techniques such as search-based software testing (SBST) and symbolic execution remains uncertain. In this paper, we perform an extensive study of automatic test generation approaches based on three tools: `EvoSuite` for SBST, `Kex` for symbolic execution, and `TestSpark` for LLM-based test generation. We evaluate tools' performance on the GitBug Java dataset and compare them using various execution-based and feature-based metrics. Our results show that while LLM-based test generation is promising, it falls behind traditional methods w.r.t. coverage. However, it significantly outperforms them in mutation scores, suggesting that LLMs provide a deeper semantic understanding of code. LLM-based approach performed worse than SBST and symbolic execution-based approaches w.r.t. fault detection capabilities. Additionally, our feature-based analysis shows that all tools are affected by the complexity and internal dependencies of the class under test (CUT), with LLM-based approaches being especially sensitive to the CUT size.**

***Index Terms*—automatic test generation, symbolic execution, concolic testing, large language models, search-based software testing**

I. Introduction

Software quality assurance is a critical aspect of the software development process, as errors in the software may lead to fatal consequences. Software testing remains one of the most widespread software quality assurance methods [3], manual testing being the most popular type of software testing [8]. Yet, automated solutions are beneficial since manual testing is a complex and time-consuming task [54]. As a result, researchers have proposed various methods for generating test cases automatically: search-based software testing (SBST) [17], [33], symbolic execution [10], concolic testing [2], etc. These techniques have advanced significantly, achieving high code coverage [24], leading to fewer smells than manually written test cases [40], and detecting unknown bugs [18]. Adoption of automated test generation tools in industry is still limited, despite these advancements [28].

Large Language Models (LLMs) offer new possibilities in software engineering, showing effectiveness in tasks like code completion [22], code understanding [37], and code generation [59]. More recently, LLMs have been applied to automatic test case generation [49], leveraging their ability to understand natural language and code context. However, LLMs face challenges such as limited context windows and hallucinations [61]. While some empirical studies suggest that LLMs are useful for test generation, they often fall short compared to traditional approaches like SBST, particularly when handling large classes [57].

We identify three critical limitations in existing comparisons between LLM-based and traditional approaches. *Limitation 1: Benchmark and data contamination*. Existing LLM-based approaches [9], [12], [21], [58], [57], [62] have been evaluated on datasets like Defects4J [21], [26] or popular GitHub projects [9], [12], which are part of the LLMs (pre)training data, introducing risks of data leakage [30], [47]. Therefore, the evaluation should be conducted using projects from different sources [47] or commits and defects discovered and fixed after the release date of the LLMs [30]. Recent works [55], [57] partially tackle this issue using the SF110 dataset or its extensions. However, this dataset has been extensively used to compare and improve SBST tools such as `EvoSuite`, thus introducing a potential positive bias towards them.

Limitation 2: No comparison with symbolic execution. SBST approaches are efficient and effective [27], [35] but they are not the only stat-of-the-art technique for generating unit tests. Symbolic execution is the first technique ever used in the literature to generate test inputs [45]. These techniques use constraint solvers to generate inputs that satisfy the conditions in the code. Still, they might struggle to satisfy conditions and large classes due to the path explosion problem [6]. Existing studies with LLMs did not consider these techniques.

Limitation 3: Lack of statistical analysis and repetitions. Existing studies do not fully account for the non-determinism of LLMs, which may produce different outputs when queried with the same prompt, or traditional test generation approaches. This issue is seldom acknowledged [57] but is yet to be addressed. In fact, LLMs have been executed only once (one seed/session), not following existing guidelines on how to assess randomized tools [4], [47] statistically.

In this paper, we conduct an extensive comparative study that addresses the three limitations discussed above. We compare three cutting-edge tools for unit test generation, namely `EvoSuite` (SBST tool), `Kex` (symbolic execution tool), and `TestSpark`. The latter tool has been configured with different LLMs: ChatGPT-4, ChatGPT-4o, Llama Medium, Code Llama 70b as we investigate the difference in performance between alternative LLMs when given with the same prompt and source code information. We use GitBug Java as the benchmark, a dataset of recent Java commits and bugs published in 2024. This dataset is designed to address data leakage issues and has not been used in prior SBST studies.

We assess each tool's performance based on execution metrics (e.g., code coverage) and feature-based metrics (e.g., complexity of the class under test). Finally, we run each tool ten times with different seeds (and independent sessions for `TestSpark`) and base our analysis on sound statistical analysis as suggested in the literature [4], [47]. Our study has two primary goals: (1) to compare the performance of LLMs, SBST, and symbolic execution in generating unit tests, and (2) to analyze the strengths and weaknesses of each approach, which could inform the development of hybrid techniques.

Overall, our results suggest that ChatGPT-4o has the best potential in automatic test generation among all LLMs. However, according to statistical tests, `TestSpark-ChatGPT-4o` still performed worse than traditional methods. We further provide insights into the characteristics of the program under tests that seem to impact the performance of the different tools, shedding light on their advantages and disadvantages.

Our main contributions can be summarized as follows:

- Open source extendable automatic test generation assessment pipeline[1].
- An extensive analysis of three automatic test generation techniques — SBST, symbolic execution-based, and LLM-based — in terms of achieved compilation rate, coverage, and mutation score of generated tests.
- A co-factor analysis between the automatic test generation techniques performance and the features of the code under test: complexity, size, language features, etc.

This paper is organized as follows. Section II gives an overview of the prior work in our area. The design and implementations of the presented test generation pipeline are described in Section III. Section IV describes all the details of the experimental setup, while V presents the evaluation results. Finally, Section VI discusses the most exciting results found during the evaluation, and Section VII discusses potential threats to validity, while Section VIII concludes the paper.

II. Related Work

A. Traditional Test Generation Approaches

Search-based software testing (SBST) [35] is one of the most popular and effective automatic test generation methods. The main idea of SBST is to use meta-heuristic optimization methods like genetic algorithms to generate test cases/suites. `EvoSuite` [17] and Pynguin [33] are two of the most popular SBST tools developed for Java and Python, respectively. They have won the SBFT/SBST unit test competitions for several years [16], [19], [25], [41].

Symbolic execution [7] is a software analysis technique that abstractly executes the target program while substituting the input values with symbolic variables. Automatic test generation is one of the main applications of symbolic execution. Concolic testing [52] is a technique that tries to solve some of the problems of symbolic execution by combining it with concrete execution. Concolic testing has been proven to be very effective in automatic test generation by tools like Klee [10], `Kex` [2], UtBot [23], etc.

Both SBST and symbolic execution have proven to be effective and reliable test generation approaches throughout the years [43]. This work focuses on Java test generation and includes evaluation of both `EvoSuite` and `Kex`.

B. LLM-Based Test Generation Approaches

`TestSpark` [48], ChatUniTest [13] and TestPilot [50] are examples of LLM-based test generation tools. TestPilot is a standalone tool for JavaScript programs, while TestSpark and ChatUniTest work with Java and provide IntelliJ IDEA plugins for better user experience. These tools are similar in their approach to test generation, which mainly consists of three steps: (1) prompt collection, (2) LLM request, and (3) a feedback loop that tries to improve the initial LLM response.

SymPrompt [46] introduces an LLM-based test generation that focuses on generating tests at the method level rather than for classes. It uses code-aware prompts to target specific paths within the method under test. While effective, this approach requires more time and incurs additional LLM requests, making it slower than other LLM-based tools. Similarly, AthenaTest [58] generates tests for individual (focal) methods. It does not include a feedback loop for the LLM, and it has been evaluated using Java static methods. We note that method-level tools have limited applicability for more complex, object-oriented scenarios involving inheritance and stateful classes.

AID [32] is a LLM-based approach that targets bug detection. This approach focuses on test generation for coding problems and stands out because it uses program specification as a part of LLM prompt and requests LLM to provide a test input generator script instead of the actual tests. This approach has shown promising results; however, it focuses on a specific use case, not general-purpose unit test generation.

Our study focuses on unit test generation in the "traditional" setting [25], where all the competitors have the same limited time budget and generate class-level tests. In that setting, LLM-based tools like `TestSpark` and ChatUniTest are the most suitable. We consider `TestSpark` for our evaluation since it can support various LLMs, such as ChatGPT, which is very common for many LLM-based test generation approaches [9], [12], [55]. Additionally, the tool can set a strict time window, automatically compile the generated tests, and integrate a feedback loop that can refine the responses of LLM.

[1]https://github.com/plan-research/tga-pipeline

Furthermore, since `TestSpark` utilizes IntelliJ IDEA in headless mode, all the context collection and code inspection features available in this IDE can be employed for prompt generation using the command line interface. As a result, we can enhance the context provided for prompt generation in our evaluation process.

C. Hybrid Test Generation Approaches

A group of approaches also combines LLM-based test generation with traditional automatic test generation methods. CodaMosa [31] is one of the first successful attempts that combined EvoSuite with LLMs. The core idea of CodaMosa is to fall back to LLM-based test generation when the search-based approach reaches its coverage plateau. CoverUp [44] is an extension of CodaMosa that improves the LLM feedback cycle by enhancing it with coverage information. However, our paper mainly focuses on studying the strengths and weaknesses of individual approaches, which can serve as the foundation for developing future hybrid strategies. Therefore, we exclude hybrid approaches from our work. Based on our findings, as future work, we plan to design new strategies to combine different tribes of AI for unit test generation and then use the existing hybrid test generation approaches as baselines.

D. Existing Comparative Studies

Recent attempts to perform a comparative study of LLM-based test generation with traditional approaches have been presented in the literature [57], [55], [21], [62]. These studies focus on evaluating the test-generating capabilities of ChatGPT 3.5 in comparison with `EvoSuite`. Tang et al. [57] also highlight the strengths and weaknesses of these two approaches. However, the studies above have clear limitations, which are also discussed in the introduction:

- All studies related to ChatGPT focus on version 3.5, which more recent versions have replaced. Besides, they do not compare different potential LLMs to use.
- These studies evaluate LLMs using either the SF10 [38], Defects-4j [26] datasets, or popular projects from GitHub. Recent studies highlight that Defects-4j and many existing GitHub projects are an integral part of the model (pre)training dataset, leading to data leakage [30], [47]. Instead, SF110 has been extensively used to assess and improve SBST tools like `EvoSuite`, introducing a potential bias toward this latter category of approaches.
- Previous studies compare only LLM-based (mostly ChatGPT-3.5) and SBST approaches.

E. Overview

Overall, we can conclude that a significant body of work is already on applying LLMs in automatic test generation. Our work, however, aims to address several limitations and focus on aspects that are not covered by the existing body of work:

- A study comparing the performance of different LLMs in the context of automatic test generation on a large-scale real-world benchmark with recent commits and changes published after the release date of used LLMs.

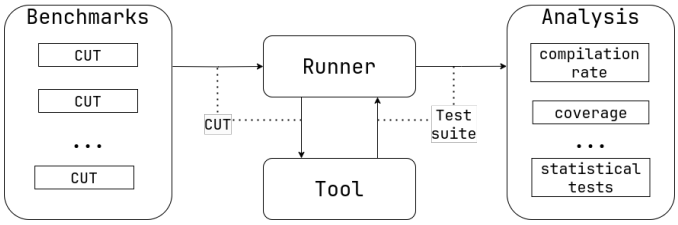


Fig. 1: Overview of the pipeline

- A comparison of the LLM-based test generation with search-based and symbolic execution-based methods.
- Analysis of strengths and weaknesses each test generation approaches regarding features of code under test.

III. Test Generation Assessment Pipeline

We have implemented a test generation pipeline to conduct an extensive study on the quality of automatic test generation for various methods. Figure 1 shows the overview of the pipeline architecture. The pipeline consists of four main parts: (1) benchmarks, (2) runner, (3) tools, and (4) analysis. The benchmark module provides a list of projects under test (PUT) in JSON format. Each PUT description includes:

- Name of the PUT;
- Version of PUT (commit hash);
- Path to the root of the project;
- Path to the sources directory;
- Path to the binaries directory;
- Full classpath;
- Name of the class under test (CUT).

The runner handles interaction with the tools and saves the results of their work, which includes:

- A path to the test sources directory;
- A list of fully qualified test names;
- A list of additional utility classes used by tests (optional);
- A list of test dependencies (e.g. JUnit, Mockito, etc.).

Each tool has a separate adapter interface that manages the interaction with the pipeline: process start, time limit, output handling, and result parsing. Currently, all the tools are executed as external CLI processes so that their execution doesn't affect the pipeline's core process. For extensibility, an adapter interface is the only requirement needed to add a new test case generation tool to our pipeline. The result of executing the runner is the test suite generated by the tool. The analysis module is executed as a post-processing. It extracts all the necessary metrics from the tools' outputs: compilation rate, coverage, mutation score, and test execution results. Then, it merges the tool's results with the pre-computed information about each benchmark (will be covered more in Section IV) and produces a CSV file with the final report. The pipeline is deployed as a swarm of Docker containers, cooperating via Docker Compose, making it easy to run in parallel.

A. Dataset

GitBug Java [56] is a reproducible benchmark of recent Java bugs published in 2024. It contains 199 bugs committed in 2023 extracted from 55 open-source repositories. This

relatively new dataset has not been included in the training set for modern LLMs, which allows us to conduct more reliable experiments. GitBug Java provides extensive information about each bug. In our study, we considered the following information: (1) project name, (2) repository URL, (3) commit hash of the buggy version, (4) commit hash of the patched version, and (5) bug patch. Unfortunately, GitBug Java does not provide explicit information about the buggy class that can be used as an automatic test generation target. Hence, for each bug, we extract information about the modified classes from the provided bug patch and manually analyze the source code to select one of them as a target (in case of any conflicts).

All of the projects from GitBug Java dataset were compiled using JDK 11, as it is the latest version of Java supported by `EvoSuite`. However, some of the projects from the dataset require newer versions of Java. After excluding these cases, our final dataset contains 136 bugs from 24 open-source repositories. The dataset includes projects of various sizes ranging from 8,000 to 300,000 source lines of code (SLOC) and CUTs of various complexity (25–2,500 SLOC) with the average cyclomatic complexity score of ≈ 66.

B. Tools

The core of our study is comparing test generation abilities of three main automatic test generation approaches: SBST, symbolic execution-based, and LLM-based. Therefore, our pipeline integrates tools from each of these categories. For SBST, we have selected `EvoSuite` [17] as it is the most well-known and proven automatic test generation tool for Java. It has shown the best results in recent editions of SBFT Java Tool Competition [25], [19], [41]. `Kex` [2] is selected as a tool that implements symbolic execution-based approach to automatic test generation, as it is the only tool using this technique that participated in the latest SBFT Java Tool Competition [25] and has shown good results in terms of coverage. For the LLM-based test generation approach, we have selected `TestSpark` [48]. There are several reasons behind this choice:

- Despite being an IntelliJ IDEA plugin, it can be easily executed in the command line by running the IDE in headless mode.
- `TestSpark` supports multiple LLM models that can be easily interchanged (fully integrated with three LLM platforms: OpenAI, HuggingFace, and JetBrains internal AI Assistant platform).
- `TestSpark` uses the powerful code inspection of IntelliJ IDEA even in the headless mode, which brings flexibility to include different contexts in the prompt.

C. Analysis

1) Execution-based metrics: These metrics are based on the compilation and execution of produced tests.

Compilation rate. The compilation rate allows us to understand the tool's reliability for automatic test generation. This metric is especially important for LLM-based tools because they often struggle with producing correct code [57]. However, most tools produce a test suite as a single source code file containing all the tests, and if there is even one syntax error, the whole test suite will not be correct. To handle these scenarios, we have extracted each test method into a separate source file. This allows us to record the compilation rate on a more granular level and also allows us to use correctly generated test methods for further analysis.

Code coverage. Coverage is one of the most used measurements of test suite quality [41]. We collected information about line, branch, and instruction coverage for each project using the JaCoCo [1] library.

Mutation score. Mutation testing [42] is a stronger test suite quality metric. Mutation score measures the "strength" of a test suite and characterizes its bug-discovering abilities. We use the PIT mutation testing system in the pipeline [14].

Failure reproduction. As the GitBug Java dataset contains a set of bugs, our pipeline evaluates the capabilities of these approaches in generating tests that can capture real-world bugs. For each entry in the dataset, we generate tests for the buggy version of the project. After, we execute tests on both buggy and patched versions of the PUT to record the differences. If there are any, it means that the tool can generate tests that are affected by the bug.

2) Code feature metrics: The quality of automatically generated tests is heavily dependent on the used approach and the code under test itself. Therefore, we perform static analysis on the CUTs in the benchmarks to extract information about distinct code features that may affect the quality of test generation. Based on that information, we can identify the strengths and weaknesses of our tools and approaches. For each CUT, we have collected the following information.

Number of dependencies. We define a dependency of CUT as an import used inside that CUT. Dependencies are categorized into internal, standard library, and external. Internal dependencies belong to the same project as CUT; standard libraries are built-in libraries in Java; external dependencies are third-party libraries not part of the previous two categories.

Comments and Java docs. Unlike the traditional automatic test generation approaches, LLM-based approaches use the source code of the CUT as the main input. Source code often includes additional contextual information in the form of comments and Java docs. These features can provide additional information about the program to LLM and thus affect the resulting tests. Additionally, we are interested not only in the presence of these features but also in the language that they are written in, as the initial instinct suggests that LLMs should perform best in the English language.

Condition types. Branching points usually add a lot of complexity to the program. However, not every branching point is equal in its complexity. Therefore, we are interested to know what types of conditions have the most impact on the quality of generated tests. We have analyzed the source code of each CUT in the dataset and extracted the information about types and sources of variables in conditions.

IV. Experimental Setup

A. Tool Setup

To address the nondeterministic nature of test generation algorithms used in this study, we repeated each execution 10 times as suggested by existing guidelines [4]. We also highlight that we applied the same methodology using multiple seeds and different sessions also for LLM-based approaches, addressing *Limitation 3* discussed in Section I.

We also need to define a time budget for our experiments as it can significantly influence the quality of generated tests [40]. In our case, we set the time budget to 120 seconds, sufficient for the test generation tools to produce meaningful tests while remaining small enough to reflect practical, everyday-use scenarios (i.e., developers' need to receive quick feedback). Furthermore, this time limit aligns with the standards used in automatic test generation competitions like SBFT [25], [19], where 120 seconds is one of the main categories.

However, most of the LLM-based approaches, including `TestSpark`, are not designed to work within the time budget. LLM-based approaches cannot modify their test generation tactic regarding time limitations, as they depend on LLM's performance: the model either successfully produces a compilable test or not. However, our preliminary experiments demonstrate that, on average, `TestSpark` takes about 2-3 minutes on each CUT, which we consider comparable to `Kex`'s and `EvoSuite`'s time budget.

Each tool we use in our experiments has many parameters that can affect its test-generation capabilities. Hence, we use each tool's default (suggested) parameters as the default parameter values commonly used in the literature give reasonably acceptable results [5] without incurring the additional computational cost required for parameter tuning.

We are using `Kex` version 0.0.8[2] in the concolic mode, and the only parameter we change is turning off built-in coverage computation. As for `EvoSuite`, we are using version 1.2.0[3]. Additionally, we are running `EvoSuite` with DynaMOSA [39] algorithm and with disabled runtime dependencies. We use DynaMOSA [39] since it has been shown to outperform other evolutionary algorithms [11], and was the default configuration in the SBFT competitions [19], [25].

LLM-based test generation introduces two more major variables into the experiment setup: (1) model selection and (2) prompt engineering. These variables introduce the following questions into LLM-based test generation setup:

- What model to choose?
- What information to include in the prompt?
- How to balance the prompt so it fits into the model's context?

As mentioned previously, `TestSpark` allows seamless switching between different LLMs. In this work, we selected the following models for the experiments: (1) ChatGPT-4, (2) ChatGPT-4o, (3) Llama Medium, and (4) Code Llama 70b.

[2]https://github.com/vorpal-research/kex/releases/tag/0.0.8

[3]https://github.com/EvoSuite/evosuite/releases/tag/v1.2.0

```
Generate unit tests in Java for $NAME to achieve
    100% line coverage for this class.
Dont use @Before and @After test methods.
Make tests as atomic as possible.
All tests should be for JUnit 4.
In case of mocking, use Mockito. But, do not use
    mocking for all tests.
Name all methods according to the template -
    [MethodUnderTest][Scenario]Test, and use only
    English letters.
The source code of class under test is as follows:
$CODE
$METHODS
$POLYMORPHISM
```

Fig. 2: Default prompt used for LLM-based test generation

This selection includes the most popular LLMs today, and one code-specific model that will allow us to compare its performance to general-purpose ones.

Listing 2 shows the default prompt used by `TestSpark`. Initially, it includes the source code of the CUT (`$CODE`), signatures of the methods accessible during the test generation (`$METHODS`), and information about polymorphic relations (`$POLYMORPHISM`). However, if the prompt turns out too big for the used model, `TestSpark` iteratively reduces its size by removing additional context information.

We are using the latest version of `TestSpark` that is available in its repository at the time of evaluation[4].

B. Research Questions

The goal of our experiments is to answer the following research questions.

- **RQ$_1$**: *What is the best LLM to use for automatic unit test generation?*
- **RQ$_2$**: *How do LLM-based automatic test generation approaches compare to traditional approaches on a large scale?*
- **RQ$_3$**: *What is the correlation between various qualities of code under test and the performance of different test generation techniques?*

To address RQ$_1$, we focus on the execution-based metrics: compilation rate, line and branch coverage, and mutation score. `TestSpark` is executed with each model under test (ChatGPT-4, ChatGPT-4o, Llama Medium, Code Llama 70b) on the GitBug Java. The performances are evaluated by analyzing the distributions of the results (metrics discussed in Section V) over different runs using boxplots and descriptive statistics (median and mean). Additionally, we perform sound statistical analysis using the Mann-Whitney U test [36] for the statistical significance, as it has already been used by prior work [25]. We further complement our analysis with the Vargha Delaney $\hat{A}_{12}$ measure [60] for effect size. These statistical tests are computed pairwise for tools for each CUT. We use p-value threshold of 0.05 to define significant differences and "small", "medium" and "large" magnitudes [60] of the $\hat{A}_{12}$ statistics to determine the winner between the two tools in the comparison on each CUT.

[4]https://github.com/JetBrains-Research/TestSpark/tree/e6adea

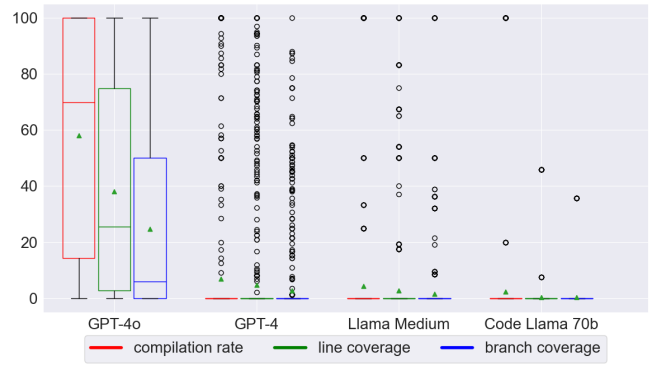


Fig. 3: Execution-metrics comparisons of the different LLMs in `TestSpark`

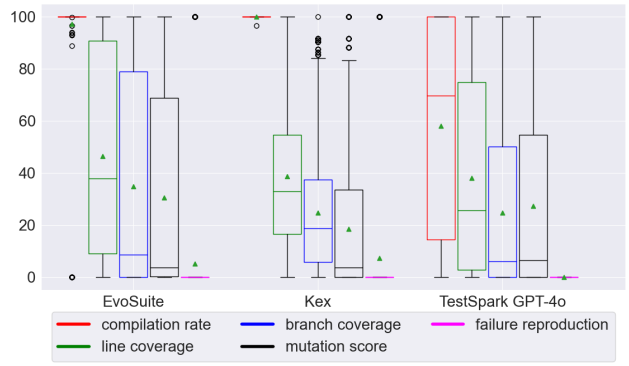


Fig. 4: Comparison of different automatic test generation tools

RQ_2 is addressed using all the same metrics mentioned for RQ_1. This time, we are comparing the results of `Kex`, `EvoSuite`, and the best performing LLM resulting from RQ_1 (which is `TestSpark-ChatGPT-4o` as shown in Section V) on the GitBug Java dataset. We also used the same statistical test for this research question. In addition to the previously mentioned metrics, we also focus on the fault reproduction capabilities of the different approaches in this experiment. We execute the generated test suite on two versions of the project: the original buggy one and the patched one. Differences in the results between these two executions means that the tool was able to capture the fault in the buggy version of the project.

RQ_3 is addressed by performing correlation analysis [20] between tool's performance and features of the CUT. CUT features are divided into two main categories:

- Correlation between coverage metrics and *code specific features* of CUT. *Code specific features* include cyclomatic complexity, number of dependencies, presence of comments and Java Docs, and SLOC.
- Correlation between coverage metrics and *branch condition types* in the CUT. *Branch conditions* are divided into the following categories: primitives, `null` operations, `switch` conditions, type checks, static/global method/variable operations, string and regex operations, collections, and others.

Analyzing the correlation between listed features of CUT and the tool's performance on a large-scale benchmark of real-world projects allows us to better understand the strengths and weaknesses of the tools in different use cases. We use Spearman's rank correlation coefficient for correlation analysis [51] as our data do not follow a normal distribution according to the Shapiro-Wilk test [53].

V. Evaluation and Results

A. RQ_1: What is the Best LLM to Use for Automatic Unit Test Generation?

Figure 3 shows the average compilation rate, line, and branch coverage for four models: ChatGPT-4o, ChatGPT-4, Llama Medium and Code Llama 70b. According to these results, ChatGPT-4o shows the best results on the benchmark, achieving an average 57.97% compilation rate, 38.13% line coverage, and 24.63% branch coverage. Other models show worse results, achieving an average of less than 7% compilation rate and 5% line coverage.

ChatGPT-4, Llama Medium and Code Llama 70b performed much worse than ChatGPT-4o. This is due to the context limitations of the latter models: while ChatGPT-4o has a 128k tokens context size, the others vary between 4-8k. Due to this, there are various instances where non-ChatGPT-4o models failed even with the smallest possible prompt size (i.e., only CUT source code). For example, out of 1360 total runs of ChatGPT-4, 1229 failed with the context size error.

In addition to the boxplots, pairwise comparison of the different LLMs using statistical tests confirms the statistical superiority of ChatGPT-4o on every measured metric. Out of $136 * 3 * 4 = 1632$ comparisons with other models, there are only two cases when ChatGPT-4o achieves statistically lower performance metrics than another model.

Due to these limitations, we conclude that ChatGPT-4o is the best viable option for automated test generation on real-world projects among the models investigated in this study. Therefore, we consider only ChatGPT-4o in the remainder of our study and further experiments.

B. RQ_2: How Do LLM-Based Automatic Test Generation Approaches Compare to Traditional Approaches on a Large Scale?

Figure 4 shows the comparison of three tools — `EvoSuite`, `Kex` and `TestSpark` with ChatGPT-4o model (`TestSpark-ChatGPT-4o` from now on) — in terms of compilation rate, line and branch coverage, mutation score and failure reproduction rate.

These graphs give us multiple insights into the performance of these tools. First, `Kex` has the best average compilation rate (99.99%) out of all three tools. Even though symbolic execution-based tools have full access to the classpath of the CUT, due to some implementation issues, `Kex` produced several uncompilable test cases. `EvoSuite` is not far from `Kex` with a 97.30% compilation rate, also caused by small implementation issues within this tool. `TestSpark-ChatGPT-4o`, as mentioned previously, has a

significantly lower compilation rate (57.97%) highlighting the unstable nature of LLM-based test generation.

In terms of coverage metrics, we can see that all three tools are close in terms of average line and branch coverage: 46.44% and 34.91% for `EvoSuite`, 38.60% and 24.72% for `Kex`, 38.13% and 24.63% for `TestSpark-ChatGPT-4o`. Although, `EvoSuite` is slightly better, especially regarding average branch coverage. However, we can see that `Kex` is significantly better in terms of median branch coverage (18.61% versus 4–6% of `EvoSuite` and `TestSpark-ChatGPT-4o`). Additionally, we can see that `Kex` performs more consistently than the other two tools.

W.r.t. mutation score, `EvoSuite` achieves the best average score (30.56%), while `TestSpark-ChatGPT-4o` was the best in terms of median score (6.32%). While the mutation scores of `EvoSuite` and `TestSpark-ChatGPT-4o` are comparable to each other, `Kex`'s performance in this metric is noticeably worse as it sometimes generates wrong oracles.

All the tools performed poorly in terms of failure reproduction of original GitBug Java bugs. `TestSpark-ChatGPT-4o` did not reproduce any bug. `EvoSuite` and `Kex` reproduced 5.88% and 7.35% of bugs, respectively. Statistical tests show that `Kex` performed better on 8 benchmarks, `EvoSuite` performed better on 6, and both tools tied on two benchmarks.

Additionally, for each CUT-metric pair, we conduct three statistical tests: `EvoSuite` vs. `Kex`, `EvoSuite` vs. `TestSpark-ChatGPT-4o`, and `Kex` vs. `TestSpark-ChatGPT-4o`. The winner between the two tools is determined based on p-value and effect size, with each tool earning 1 point for winning a statistical test. The tool(s) with the highest final scores for each CUT-metric pair are considered the winners. Figure 5 visualizes these comparisons with Venn diagrams, categorizing each CUT based on the winning tool(s) in terms of line coverage (5a), branch coverage (5b), and mutation score (5c). We can see that `EvoSuite` performed best in terms of line and branch coverage, closely followed by `Kex`. `TestSpark-ChatGPT-4o` falls behind in both metrics, but it performs noticeably better on the mutation score comparison, closely followed by `EvoSuite`. `Kex` shows the worst performance on the mutation score.

In summary, we conclude that the tools are comparable regarding coverage-based metrics. `EvoSuite` and `Kex` excel more at line and branch coverage, while `TestSpark-ChatGPT-4o` shows better results in mutation score. W.r.t. to fault detection capability, the test generated by `EvoSuite` and `Kex` outperformed `TestSpark-ChatGPT-4o`, which failed to detect any fault.

C. RQ_3: What is the Correlation between Various Qualities of Code Under Test and the Performance of Different Test Generation Techniques?

Figure 6 presents the results of the correlation analysis between the coverage-based metrics of the tools and the code-specific features of the CUT. We used Spearman's rank correlation coefficient to measure the strength and direction of these relationships. Correlations were classified as weak (0.00–0.30), moderate (0.31–0.60), or strong (0.61–1.00), with the sign indicating positive or negative correlations. These cut-off values follow the suggested classification guidelines [15].

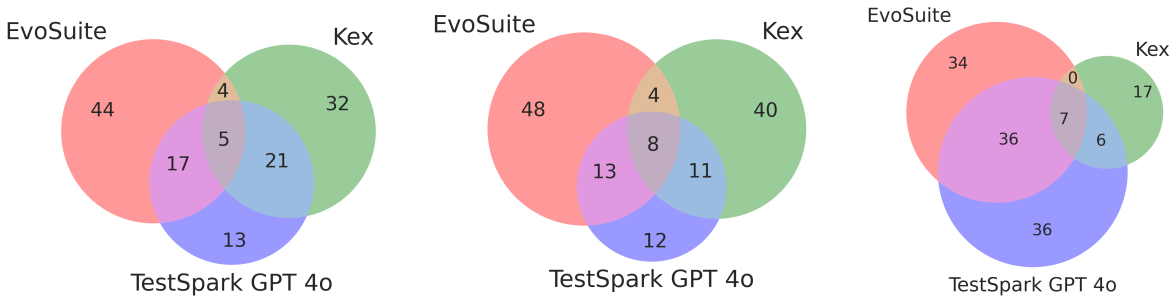
We notice that all tools are significantly impacted by three features: the cyclomatic complexity of the CUT, the number of its internal dependencies, and SLOC. However, while the coverage of all the tools is affected approximately similarly by cyclomatic complexity, there is more variation in correlation with the number of dependencies and with SLOC. `Kex` shows the lowest correlation with these features, `EvoSuite` shows a higher correlation with the number of internal dependencies, and `TestSpark-ChatGPT-4o` shows a strong correlation with both the numbers of dependencies and SLOC.

Surprisingly, all the tools show a weak correlation between their coverage performance and the number of Java standard library dependencies in the CUT, even though these are some of the most common dependencies in Java. Additionally, we can see that `EvoSuite` and `Kex` have a strong negative correlation with the number of external dependencies of the project, while `TestSpark-ChatGPT-4o` shows a weak correlation (i.e., lower than 30%).

Additionally, `TestSpark-ChatGPT-4o` shows a weak correlation with the presence of comments and Java Docs in the source code: comments seem to slightly worsen the test generation capabilities of ChatGPT-4o (-0.17), while Java Docs seems to have a weak positive correlation with the coverage-related metrics (0.1). `Kex` and `EvoSuite` do not correlate with these features. This result is expected since both tools work on the bytecode level (which does not include comments and Java Docs) and cannot access source code.

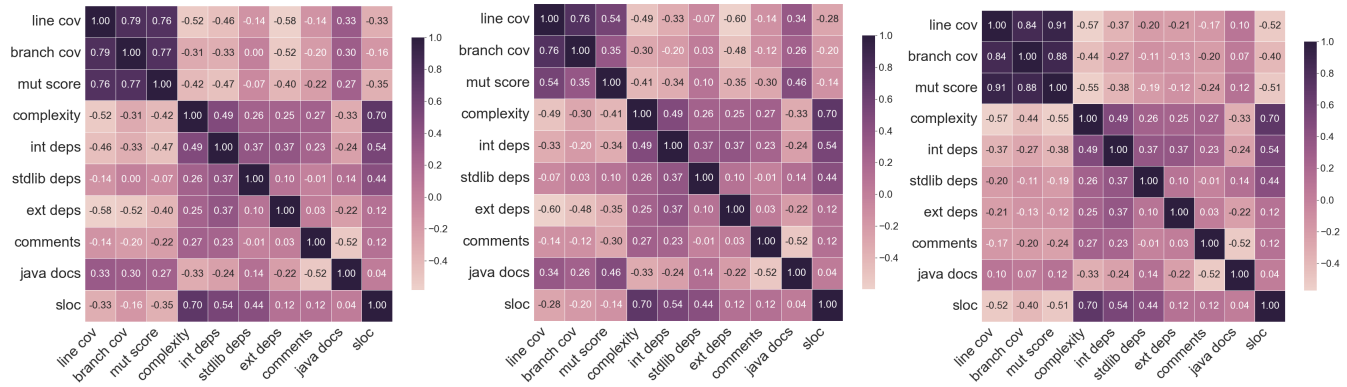
Figure 7 shows a correlation analysis between coverage-based metrics of the tools and branch types of the CUT. Overall, all the tools show almost no positive correlation with any of the branch types. According to Figure 7a, `EvoSuite` shows a negative correlation with several branch types. However, the switch conditions, static methods, and standard library calls negatively correlate to its performance. Additionally, we can see that coverage and mutation score achieved by `EvoSuite` positively correlates with the number of type checks in the CUT. It is worth noticing that the presence of `null` checks has a moderate negative correlation with line coverage and mutation score. Null checks are a well-known issue in SBST as they constitute the so-called flag problem [34], [35]: an object is either null or not, and the existing heuristics (namely approach level [34] and branch distance [29]) do not provide any guidance toward satisfying these checks.

Figure 7b shows `Kex`'s correlation analysis regarding coverage metrics. As `EvoSuite`, `Kex`'s performance has a strong negative correlation with the number of switch statements and static calls in the CUT. Unlike `EvoSuite`, `Kex` is much less affected by `null` checks, string operations, and standard library calls. Additionally, `Kex` shows a negative correlation with the number of primitive type conditions and collections in CUT. Moreover, we see that `Kex` has a strong positive



(a) Line coverage (b) Branch coverage (c) Mutation score

Fig. 5: Venn diagrams of tools performances across CUTs, computed using pairwise comparisons of tools using p-value and effect size



(a) `EvoSuite` (b) `Kex` (c) `TestSpark-ChatGPT-4o`

Fig. 6: Correlation analysis of tool performance and code-specific features

correlation with the number of type conditions in the CUT.

Figure 7b presents the results of the correlation analysis of `TestSpark-ChatGPT-4o`'s performance. Overall, `TestSpark-ChatGPT-4o` shows a much higher negative correlation with most branch types encountered in the CUT.

VI. Discussion

Regarding RQ_1, the evaluation results give a definitive answer: ChatGPT-4o is the best LLM (among the ones we tested) for application in automatic test generation. As mentioned before, the main factor that played a role in this experiment is the context size of the model. Real-world programs are too big and too complex for the smaller LLMs. Manual analysis of CUTs that were unaffected by context size limitations further shows that the smaller models perform slightly worse than ChatGPT-4o. The three projects `traccar` (commit `074dc016d2`), `cbor-java` (commit `b010e8c62b`) and `semver4j` (commit `48ffbfd1f6`) are stand out as clear instances of the observed difference in performance.

RQ_2 has no single definitive answer. `Kex` and `EvoSuite` show very comparable performance on the line and branch coverage metrics, while `TestSpark-ChatGPT-4o` falls behind. However, `TestSpark-ChatGPT-4o` shows a better median (but not mean) mutation score than the other two tools. This seems to suggest ChatGPT-4o might have a better understanding of the expected behavior of CUT and, therefore, potentially generate better oracles. However, this observation holds only for ChatGPT-4o while the other LLMs struggle to generate compilable test cases.

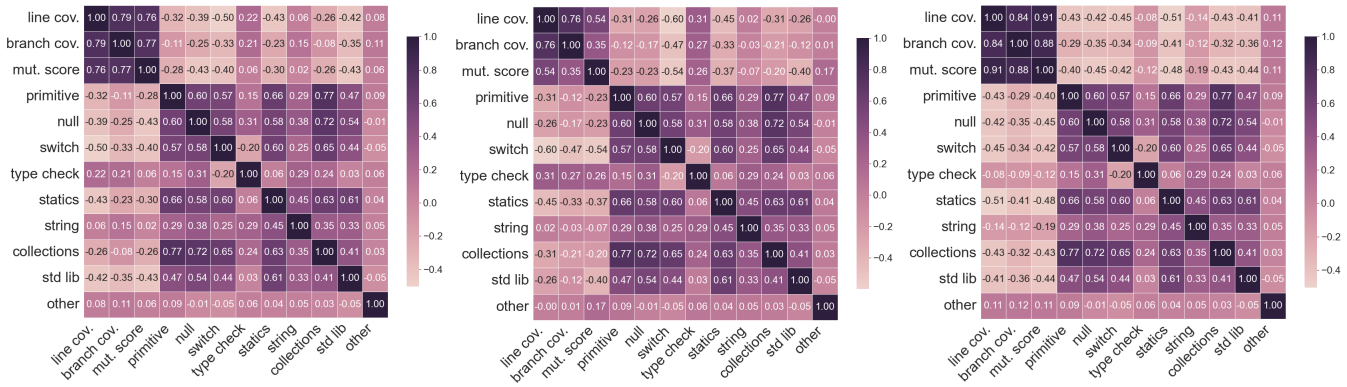
Finally, all tools struggle with fault detection capability. However, `TestSpark-ChatGPT-4o` could not detect any fault in our experiments' runs. While the other two tools were able to seldom expose some faults in the benchmark.

Evaluation results demonstrate that each approach has a set of use cases where it outperforms others. Thus, the best possible automatic test generation tool should incorporate all these approaches to achieve better results. Our Venn diagrams highlight this complementarity.

As an example, the Venn diagrams (figure 5) demonstrate that `EvoSuite` and `Kex` have a very small intersection across all three metrics, meaning that these two tools (and, therefore, approaches) excel at different CUTs. Moreover, `Kex` has an even smaller intersection with both tools in the mutation scores, meaning that, even though it generates worse oracles in general, in some cases, its oracles are unique in comparison with `EvoSuite` and `TestSpark-ChatGPT-4o`.

Manual analysis. We manually analyzed the subset of 32 benchmarks that demonstrated significant differences in tool performances. Although it is hard to generalize beyond our benchmark and dataset, we can provide valuable insights:

- `EvoSuite` and `TestSpark-ChatGPT-4o` are very effective at generating tests for string-



(a) `EvoSuite` (b) `Kex` (c) `TestSpark-ChatGPT-4o`

Fig. 7: Correlation analysis of tool performance and branch features

processing CUTs. For example, benchmark `jsoup` (commit `e1880ad73e`) is a URL builder class; `EvoSuite` and `TestSpark-ChatGPT-4o` are able to generate valid URL strings in their tests. However, when it comes to more complex and non-standard formats, `TestSpark-ChatGPT-4o` can outperform `EvoSuite` (e.g., the `semver4j` commit `48ffbfd1f6`).

- `TestSpark-ChatGPT-4o` is often let down by the compilation errors of the generated tests. Benchmarks like `Simple-DSL` (commit `81182e58bd`) and `traccar` (commit `28440b7726`) show that when all the tests compile, `TestSpark-ChatGPT-4o` can achieve on par or even better coverage than `Kex` and `EvoSuite`.
- `EvoSuite` and `TestSpark-ChatGPT-4o` use mocking when working with interfaces and standard library classes. `Kex` only uses mocks when working with abstract classes. Benchmarks like `dataframe-ec` (commit `1109752c4e`) and `traccar` (commit `596036dc33`) show how mocking allows `EvoSuite` and `TestSpark-ChatGPT-4o` outperform `Kex`.
- Our manual analysis confirmed that `TestSpark` with ChatGPT-4o generates more and better oracles in many cases. For example, on `traccar` (commit `596036dc33`), both `EvoSuite` and `Kex` failed to generate oracles for some of their tests.
- `Kex` often outperforms other tools on a very complex CUTs with the complex arguments. Benchmarks like `graphql-java-annotations` (commit `6d9d7a79de`), `jsoup` (`b6f652cef6`), and `frigga` (commit `6b520bbb2e`) highlight that. Additionally, `Kex` outperforms other tools on benchmarks that contain some bit-level operations and conditions (e.g. `traccar-2be2a4558a`).
- Our results suggest that the 120-second time limit is not always enough for the intensive approach of `Kex`. Benchmarks `java-solutions-7a73ea56d0` and `cbor-java-b010e8c62b` are instances that led us to this conclusion.

Correlation analysis. Feature-based correlation analysis also highlights several insights into automatic test generation tools' performance. First, each tool demonstrates moderate dependency on the CUTs SLOC and computational complexity. Interestingly, `TestSpark-ChatGPT-4o` is more dependent on the SLOC than `Kex` and `EvoSuite`, meaning that LLM-based models are better for smaller classes not only because of the context limitations. Secondly, the correlation analysis demonstrates that all the tools have a moderate correlation with the number of internal and external dependencies of CUT. `TestSpark-ChatGPT-4o` is more affected by internal dependencies, meaning that it is hard for LLM-based methods to work with closely interconnected projects. `EvoSuite` and `Kex` are expectedly more affected by the number of external dependencies, as their number heavily affects the complexity of the analysis.

We also focus on evaluating how the additional information within the code (like comments and Java Docs) can affect the LLMs performance. We do not see any strong correlation between these features and the tool's performance, but still, we can see that comments in code have a weak negative correlation (-0.17), and Java Docs demonstrate a weak positive correlation (0.1). Additionally, it is worth noting that 135 of 136 projects used the English language in the code for variable and class names, comments, docs, etc. In additional experiments, where we translated the CUTs to Spanish using ChatGPT-4o, no significant performance variations were observed across different languages. Due to space constraints, further details of this analysis are not included in the paper.

Branch type-based correlation analysis of the tool's results also demonstrates some interesting results. Firstly, all tools demonstrate a negative correlation with almost all branching types, meaning that, expectedly, any branch complicates things for the test generation. Secondly, all tools demonstrate a moderate negative correlation with the number of primitive and switch conditions in the CUT. Although it is unexpected at first, manual analysis of the results shows that it is mainly because of the following:

- Sheer number of such conditions; especially switch cases usually contain many branching points;
- The source of the variables in these conditions: in many

cases, these variables are products of some complicated operations (e.g., file reads, collection interactions, etc.).

Third, we can see that `TestSpark-ChatGPT-4o` shows a moderate negative correlation with most branch types except strings and type checks. Manual analysis results confirm that trend: LLM-based test generation shows promising results with string processing CUTs and struggles more with the others.

Finally, only `Kex` and `EvoSuite` show a weak positive correlation with the number of type checks in the code. It is the only condition type that positively correlates with any tool's performance. While not obvious, this fact can be easily explained. Type checks are usually relatively easy for these tools (especially for `Kex`), as they are easily satisfiable and do not have a lot of options. At the same time, type checks provide the tool with additional contextual information and simplify the further generation.

VII. Threats to Validity

In this section, we acknowledge the threats that may affect the validity of our experimental results.

Internal threats: We ensured the correctness of every implementation step and manually analyzed the portions of the result to ensure their correctness. However, it is still possible that our implementation contains some bugs and issues.

Additionally, the difference in the time budget handling for LLM-based and traditional test generation approaches can introduce threats to the validity of our results. It is hard to compare these approaches fairly because of their unique features. Our experiment results suggest that the time budget handling did not substantially affect the quality of our results.

External threats: LLMs introduces many data-related threats to the computer science research. Data leakage is one of the most important ones. Because of the nature of LLM training, modern LLMs have been exposed to many open-source software projects during the training process. Thus, they may demonstrate significant differences in test generation quality on the projects they have seen during (pre)training and those they have not seen. Even though GitBug Java dataset was published later than the alleged training time of all the LLMs used in our experiments, the projects used in GitBug Java were still available for LLMs for (pre)training.

Another potential threat is the dataset itself. It contains a lot of different projects and bugs; however, it may not be representative enough. Moreover, the GitBug Java dataset has a small imbalance in the project distribution. Out of the 136 bugs used in our evaluation, 29 were related to `jsoup` project, and 70 were related to `traccar` project.

Conclusion threats: A potential threat to validity is related to the non-determinism of LLMs, `EvoSuite`, and `Kex`. To address this potential threat (and the limitation of existing comparative studies involving LLMs), we run each tool ten times of each CUT in our benchmark. Besides, for LLMs, we use different/separate sessions for promoting/queries to ensure that the model did not learn from past interactions and prompts. For our analysis, we compare the results based on the median and mean results achieved w.r.t. well-established test quality metrics (i.e., coverage, mutation score, and fault detection capability) and relied on sound statistical analysis, following existing guideliens [4], [47]. More specifically, we have used non-parametric tests that do not make any assumption on the data distributions being compared, namely the Mann-Whitney U Tests (or the Wilcoxon rank sum test), the Vargha-Delaney $\hat{A}_{12}$ statistics, and the Spearman's rank correlation coefficient.

VIII. Conclusion

In this paper, we present the results of our extensive evaluation of three automatic test generation approaches: SBST, symbolic execution, and LLM-based test generation. We evaluated three tools that implement the aforementioned approaches — `EvoSuite`, `Kex`, and `TestSpark` respectively — on the bugs from GitBug Java dataset with the main focus of highlighting the strengths and weaknesses of each approach.

Our evaluation results demonstrate that LLM-based test generation approaches are very heavily reliant on the context size of the LLM: 4-8k context size is not big enough for real-world applications. The best performing LLM out of the four tested is ChatGPT-4o.

`EvoSuite` and `Kex` perform very close in terms of coverage-based metrics like line and branch coverage, while `TestSpark-ChatGPT-4o` falls slightly behind. However, the LLM-based approach shows a noticeable improvement in the mutation score, suggesting that LLMs are more capable of deeper code understanding. Our evaluation also revealed that LLM-based performed worse than traditional approaches in terms of fault detection capabilities and failed to reproduce anything. Manual analysis results additionally highlighted some of the differences in tool performances.

We plan to create a new multi-approach test generation tool in future work. Our evaluation results suggest that we can find the best approach for each CUT and combine the strengths of different approaches.

IX. Data availability

The reproduction package with the dataset collection scripts and the results presented in the evaluation is available at https://doi.org/10.5281/zenodo.13862019.

X. Acknowledgements

This work was conducted as part of the AI for Software Engineering (AI4SE) collaboration between JetBrains and Delft University of Technology. The authors gratefully acknowledge the financial support provided by JetBrains, which made this research possible.

REFERENCES

[1] Jacoco — java code coverage library. [Online]. Available: https://www.jacoco.org/jacoco/trunk/index.html

[2] A. M. Abdullin and V. Itsykson, "Kex: A platform for analysis of jvm programs," *Information and control systems*, no. 1 (116), pp. 30–43, 2022.

[3] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2017.

[4] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[5] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, pp. 594–623, 2013.

[6] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[7] ——, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[8] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 179–190. [Online]. Available: https://doi.org/10.1145/2786805.2786843

[9] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, "Unit test generation using generative ai: A comparative performance analysis of autogeneration tools," in *Proceedings of the 1st International Workshop on Large Language Models for Code*, 2024, pp. 54–61.

[10] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[11] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Information and Software Technology*, vol. 104, pp. 207–235, 2018.

[12] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 572–576.

[13] ——, "Chatunitest: A framework for llm-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 572–576.

[14] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 449–452.

[15] W. Conover, *Practical nonparametric statistics*. John Wiley & Sons, Inc, 1999.

[16] N. Erni, M. Al-Ameen, C. Birchler, P. Derakhshanfar, S. Lukasczyk, and S. Panichella, "Sbft tool competition 2024-python test case generation track," in *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, 2024, pp. 37–40.

[17] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[18] ——, "1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite," *Empirical software engineering*, vol. 20, pp. 611–639, 2015.

[19] A. Gambi, G. Jahangirova, V. Riccio, and F. Zampetti, "Sbst tool competition 2022," in *Proceedings of the 15th Workshop on Search-Based Software Testing*, 2022, pp. 25–32.

[20] N. J. Gogtay and U. M. Thatte, "Principles of correlation analysis," *Journal of the Association of Physicians of India*, vol. 65, no. 3, pp. 78–81, 2017.

[21] S. Gu, C. Fang, Q. Zhang, F. Tian, and Z. Chen, "Testart: Improving llm-based unit test via co-evolution of automated generation and repair iteration," *arXiv preprint arXiv:2408.03095*, 2024.

[22] D. Guo, C. Xu, N. Duan, J. Yin, and J. McAuley, "Longcoder: A long-range pre-trained language model for code completion," in *International Conference on Machine Learning*. PMLR, 2023, pp. 12 098–12 107.

[23] D. Ivanov, A. Menshutin, D. Fokin, Y. Kamenev, S. Pospelov, E. Kulikov, and N. Stroganov, "Utbot java at the sbst2022 tool competition," in *Proceedings of the 15th Workshop on Search-Based Software Testing*, 2022, pp. 39–40.

[24] G. Jahangirova and V. Terragni, "Sbft tool competition 2023-java test case generation track," in *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE, 2023, pp. 61–64.

[25] ——, "Sbft tool competition 2023-java test case generation track," in *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE, 2023, pp. 61–64.

[26] R. Just, "Defects4j-a database of real faults and an experimental infrastructure to enable controlled experiments in software engineering research," 2019.

[27] M. Khari and P. Kumar, "An extensive evaluation of search-based software testing: a review," *Soft Computing*, vol. 23, pp. 1933–1946, 2019.

[28] C. Klammer and R. Ramler, "A journey from manual testing to automated test generation in an industry project," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2017, pp. 591–592.

[29] B. Korel, "Automated software test data generation," *IEEE Transactions on software engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[30] J. Y. Lee, S. Kang, J. Yoon, and S. Yoo, "The github recent bugs dataset for evaluating llm-based debugging applications," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2024, pp. 442–444.

[31] 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 *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.

[32] K. Liu, Y. Liu, Z. Chen, J. M. Zhang, Y. Han, Y. Ma, G. Li, and G. Huang, "Llm-powered test case generation for detecting tricky bugs," *arXiv preprint arXiv:2404.10304*, 2024.

[33] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.

[34] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[35] ——, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 153–163.

[36] N. Nachar *et al.*, "The mann-whitney u: A test for assessing whether two independent samples come from the same distribution," *Tutorials in quantitative Methods for Psychology*, vol. 4, no. 1, pp. 13–20, 2008.

[37] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[38] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[39] ——, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[40] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Test smells 20 years later: detectability, validity, and reliability," *Empirical Software Engineering*, vol. 27, no. 7, p. 170, 2022.

[41] S. Panichella, A. Gambi, F. Zampetti, and V. Riccio, "SBST tool competition 2021," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2021, pp. 20–27.

[42] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[43] I. Papadhopulli and N. Frasheri, "Todays challenges of symbolic execution and search-based for automated structural testing," in *Proceeding of the 4th Internetional Virtual Conference*, 2015, pp. 23–27.

[44] J. A. Pizzorno and E. D. Berger, "Coverup: Coverage-guided llm-based test generation," *arXiv preprint arXiv:2403.16218*, 2024.

[45] C. V. Ramamoorthy, S.-B. Ho, and W. Chen, "On the automated generation of program test data," *IEEE Transactions on software engineering*, no. 4, pp. 293–300, 1976.

[46] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 951–971, 2024.

[47] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 102–106.

[48] A. Sapozhnikov, M. Olsthoorn, A. Panichella, V. Kovalenko, and P. Derakhshanfar, "Testspark: Intellij idea's ultimate test generation companion," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 30–34.

[49] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, 2023.

[50] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," 2023. [Online]. Available: https://arxiv.org/abs/2302.06527

[51] P. Sedgwick, "Spearman's rank correlation coefficient," *Bmj*, vol. 349, 2014.

[52] K. Sen, "Concolic testing," in *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, 2007, pp. 571–572.

[53] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3-4, pp. 591–611, 1965.

[54] R. Sharma, "Quantitative analysis of automation and manual testing," *International journal of engineering and innovative technology*, vol. 4, no. 1, 2014.

[55] M. L. Siddiq, J. C. Da Silva Santos, R. H. Tanvir, N. Ulfat, F. Al Rifat, and V. Carvalho Lopes, "Using large language models to generate junit tests: An empirical study," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 313–322.

[56] A. Silva, N. Saavedra, and M. Monperrus, "Gitbug-java: A reproducible benchmark of recent java bugs," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024.

[57] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, "Chatgpt vs sbst: A comparative assessment of unit test suite generation," *IEEE Transactions on Software Engineering*, 2024.

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

[59] S. Ugare, T. Suresh, H. Kang, S. Misailovic, and G. Singh, "Improving llm code generation with grammar augmentation," *arXiv preprint arXiv:2403.01632*, 2024.

[60] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[61] J.-Y. Yao, K.-P. Ning, Z.-H. Liu, M.-N. Ning, and L. Yuan, "Llm lies: Hallucinations are not bugs, but features as adversarial examples," *arXiv preprint arXiv:2310.01469*, 2023.

[62] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1703–1726, 2024.