E-Test: E'er-Improving Test Suites

Ketai Qiu
Università della Svizzera Italiana (USI)
Lugano, Switzerland
ketai.qiu@usi.ch

Luca Di Grazia
Università della Svizzera Italiana (USI)
Lugano, Switzerland
work@lucadigrazia.com

Leonardo Mariani
Università Milano-Bicocca
Milan, Italy
leonardo.mariani@unimib.it

Mauro Pezzè
Università della Svizzera Italiana (USI)
Lugano, Switzerland; Università Milano-Bicocca
Milan, Italy; Constructor Institute of Technology (CIT)
Schaffausen, Switzerland
mauro.pezze@usi.ch

Abstract

Test suites are inherently imperfect, and testers can always enrich a suite with new test cases that improve its quality and, consequently, the reliability of the target software system. However, finding test cases that explore execution scenarios beyond the scope of an existing suite can be extremely challenging and labor-intensive, particularly when managing large test suites over extended periods.

In this paper, we propose E-Test, an approach that reduces the gap between the execution space explored with a test suite and the executions experienced after testing by augmenting the test suite with test cases that explore execution scenarios that emerge in production. E-Test (i) identifies executions that have not yet been tested from large sets of scenarios, such as those monitored during intensive production usage, and (ii) generates new test cases that enhance the test suite. E-Test leverages Large Language Models (LLMs) to pinpoint scenarios that the current test suite does not adequately cover, and augments the suite with test cases that execute these scenarios.

Our evaluation on a dataset of 1,975 scenarios, collected from highly-starred open-source Java projects already in production and Defects4J, demonstrates that E-Test retrieves not-yet-tested execution scenarios significantly better than state-of-the-art approaches. While existing regression testing and field testing approaches for this task achieve a maximum F1-score of 0.34, and vanilla LLMs achieve a maximum F1-score of 0.39, E-Test reaches 0.55.

These results highlight the impact of E-Test in enhancing test suites by effectively targeting not-yet-tested execution scenarios and reducing manual effort required for maintaining test suites.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

ICSE 2026, April 2026, Rio De Janeiro, Brazil

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Ketai Qiu, Luca Di Grazia, Leonardo Mariani, and Mauro Pezzè. 2026. E-Test: E'er-Improving Test Suites. In *Proceedings of 48th International Conference on Software Engineering (ICSE 2026).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Test suites are extremely important to guarantee the quality of software systems [27, 69]. Test suites are inherently imperfect, that is, they never reveal all bugs in the software system, and can always be improved, no matter how well maintained they are [29, 32]. Even mature software systems undergo continuous testing [1, 3, 41]. An effective quality process of a long-lasting software product requires test suites that software engineers maintain and evolve throughout the software lifetime [25, 50, 51, 61].

So far the research about long-lasting testing has focused mostly on test augmentation, regression and field testing [12, 16, 33]. Test augmentation and regression approaches remove obsolete test cases, select, prioritize and filter test cases to control the size of the suite [24, 28, 36, 39, 47, 67], and augment the suite with test cases that exercise the modified code [54]. Field testing executes software systems in the field to reveal failures that emerge only in production [9, 23], usually executing test cases that instantiate templates in the production environment [4].

In this paper we tackle the problem of long-lasting testing from a different and more ambitious perspective than regression and field testing. We propose *e'er-improving test suites*[1], test suites that automatically improve with data that become available at any time during the software lifecycle (a major source being data from monitoring the execution of the software system in production) to widen the executions that the test suite exercises, thus increasing the coverage of execution scenarios. We start from the observation that the execution in production is by far the largest and most complete set of actual execution scenarios of a software system [7]. We observe that the execution scenarios from production offer a unique opportunity to *explore not-yet-tested scenarios*. Unfortunately, simply augmenting the test suite with all the scenarios observed in the software lifetime blows up the suite that quickly becomes useless [20]. We argue that it is indeed possible to continuously improve a test suite, by sifting huge sets of scenarios, like the ones that we obtain

[1] *E'er* is an ancient English term for *Ever*.

from monitoring the execution in production, to identify *need-test* and *error-prone* scenarios, and generate test cases from them.

We propose E-Test, an approach that classifies execution scenarios as *already-tested*, *need-test* and *error-prone* with respect to a test suite, and enriches the test suite with new test cases that exercise *need-test* and *error-prone* scenarios. An execution scenario for a Software Under Test (SUT) contains the input and the state for the SUT. The scenario is *already-tested* with respect to a test suite, if the input is *"equivalent"* to a test case in the suite. It is *need-test*, if there is *"no equivalent test"* case in the suite and it succeeds, *error-prone* if the method fails.

Classifying scenarios by executing all the test cases is impractical for industrial-scale software and test suites. E-Test efficiently classifies scenarios without executing the test cases. Our assumption is that Large Language Models (LLMs) can effectively classify scenarios without executing all test cases thanks to the huge corpus of public bug reports, issues, code, and tests that belong to the training data of the LLM. However, the use of vanilla LLMs does not produce the expected results, as we report in Section 3. E-Test fine tunes an LLM, defines a set of prompts, queries the LLM with Retrieval-Augmented Generation (RAG) and merges the results of querying the LLM, to overcome the limitations and improve the poor effectiveness of a vanilla LLM.

E-Test effectively classifies the execution scenarios, by investigating four key characteristics that characterize the effectiveness of a scenario with respect to a test suite: the ability to improve *coverage*, the *uniqueness of the execution* of the scenario, the *correctness* of the output, and the likelihood of the scenario to reveal a *bug*. We carefully engineered a prompt template that assesses an input execution scenario, by asking five questions about the four characteristics, and we defined a strategy that combines the answers to the questions, to effectively classify the scenario as *already-tested*, *need-test* or *error-prone*.

We created a dataset of 1,975 scenarios from both popular Java projects available on GitHub and extending Defects4J [30]. We used the dataset to experimentally evaluate both the effectiveness of E-Test to classify execution scenarios and the sensitivity of E-Test to different configurations[2]. The experimental results indicate that E-Test achieves 0.55 precision, 0.59 recall, 0.55 F1-score, on average. We compared E-Test to state-of-the-art approaches for both regression testing and field testing, *FAST++* [14] and *field-ready testing* [22], respectively. The comparative evaluation indicates that E-Test outperforms both *FAST++* and *field-ready testing* in terms of F1-score, with a relative increase of 61.8% with respect to *FAST++* and 83.3% with respect to *field-ready testing*. We automatically generated test cases for all Defects4J scenarios that E-Test classifies as *error-prone*, to check for the effectiveness of E-Test to generate error-revealing test cases. The generated test cases detect 83.2% of the failures documented for the considered scenarios in Defects4J.

This paper contributes to state of the art in software testing by:

- spotlighting a novel viewpoint of *e'er-improving test suites*: improving test suites by identifying execution scenarios that can enrich the suite,
- defining E-Test, an approach that leverages LLMs with prompts, RAG, fine-tuning, and merging of the results of multiple queries, to classify scenarios as *already-tested*, *need-test*, or *error-prone*, and thus to identify new candidate test cases,
- creating a dataset of 1,975 execution scenarios that we harvested from both Defects4J and four popular Java projects available on GitHub, to evaluate E-Test,
- discussing the results of a thorough experimental evaluation of E-Test across different Java projects, for comparatively evaluating E-Test with respect to vanilla LLMs and state-of-the-art regression and field testing approaches,
- presenting a test case generator from execution scenarios, and evaluating the ability of E-Test to reveal failures.

The paper is organized as follows. Section 2 defines E-Test and introduces a running example. Section 3 discusses the experimental results. Section 4 overviews related work. Section 5 summarizes the main results and spotlights the open research directions.

2 E-Test

E'er-improving test suites are long-lasting test suites that automatically grow with new test cases that incrementally explore new partitions of the execution space, that is, exercise behaviors that the original test suite misses. E-Test automatically identifies test cases that exercise unexplored behaviors from sets of execution scenarios, by classifying scenarios as *already-tested*, *need-test* or *error-prone*. When fed with execution scenarios from production, E-Test efficiently identifies behaviors that are observed in production and not well-tested yet. Thus, it reduces the gap between the tested and production spaces, that is, the executions that the test suite exercises and the executions in production.

E-Test works with a SUT, a test suite for the SUT, and the input for the SUT, and indicates whether the input exercises a new behavior and thus is a good candidate to generate a test case that improves the test suite. The definition of E-Test does not restrict the application of the approach to a specific testing level. In principle, E-Test can be instantiated for unit, integration and system testing. In this paper, we present E-Test for augmenting unit test suites, to benefit from large and widely used benchmarks for fine-tuning the core LLM component of E-Test. Adapting E-Test to integration and system testing requires suitable benchmarks for fine-tuning.

An execution scenario for a method is a tuple $\langle In, M, T \rangle$, where In is an input for a method M, M is a method with its *context data*, and T is a test suite for M. The context data for a method M are the *operations executed to build the objects* that method M uses as parameters and accesses as external variables, if any. For example, $\mathcal{S}$ is an execution scenario for the method `asNode` in `Spring Boot`[3]:

$$\mathcal{S} = \langle \texttt{"[::1]:6379"},\ \texttt{asNode},\ \mathcal{T} \rangle$$

$$\mathcal{T} = \{\texttt{asNode("127.0.0.1:1111")},\ \texttt{asNode("127.0.0.2:2222")},\ \texttt{asNode("127.0.0.3:3333")}\}$$

where `"[::1]:6379"` is the input and $\mathcal{T}$ is the test suite for the method `asNode`. We use this scenario that led to a failure in production (GitHub issue #39819[4], diagnosed on March 1, 2024) as a running example in this paper.

[2] E-Test is available on a replication package at https://github.com/ketaiq/E-Test-package to facilitate independent replicas of the experiments.

[3] https://github.com/spring-projects/spring-boot

[4] https://github.com/spring-projects/spring-boot/pull/39819

An execution scenario $\langle In, M, T\rangle$ is *already-tested*, if In is "*equivalent*" to a test in T. Two test cases are equivalent if they belong to the same partition, from the partition testing viewpoint [53]. We recall that partition testing *"divides the infinite set of possible test cases into a finite set of classes (partitions), with the purpose of drawing one or more test cases from each class"* and that each partition groups test cases with uniform behavior, that is, test cases that exercise the same portion of the program, according to some test case selection criterion [49]. The identification of partitions depends on the test case selection criterion. In our experiments, we refer to a combination of structural and functional criteria that lead to the definition of the set of heterogeneous queries.

An execution scenario $\langle In, M, T\rangle$ is either *need-test* or *error-prone*, if In is *"not equivalent"* to a test in T for M, and M either succeeds or fails when executed with In, respectively. *Already-tested* scenarios increase the size of the suite without enriching it. *Need-test* scenarios enrich the test suite by sampling not-yet-tested partitions, thus augmenting the quality of the test suite. *Error-prone* scenarios reveal bugs to be pruned from the code.

The example scenario $\mathcal{S}$ executes the method `asNode` with an input IPv6 address (`"[::1]:6379"`), an IP format that is not exercised with the test cases in $\mathcal{T}$ that use the IPv4 format (`x.x.x.x:xxxx`). The method `asNode` fails in this scenario, due to a parsing error, as documented in the GitHub issue. Thus, the scenario $\mathcal{S}$ is *error-prone*, since it exercises a scenario different from the scenarios that the suite exercises, and causes the method to fail. The scenario ⟨`"127.0.0.1:5678"`, `asNode`, $\mathcal{T}$⟩ is *already-tested* with respect to the test suite $\mathcal{T}$, since it exercises executions that the tests in $\mathcal{T}$ already exercised, and the method `asNode` does not fail with the input `"127.0.0.1:5678"`. We observe that the scenario $\mathcal{S}$ becomes *need-test* for the method `asNode` after fixing the bug, since it exercises an execution that the tests in $\mathcal{T}$ do not exercise, and the fixed method does not fail.

Figure 1 illustrates the three phases of E-Test: PreProcessor, Analyzer, and PostProcessor. The PreProcessor builds the prompts for the Analyzer from the execution scenario. The Analyzer queries an LLM to answer the prompt. The PostProcessor classifies the input scenario as *already-tested*, *need-test*, or *error-prone*, and generates test cases for both *need-test* and *error-prone* scenarios.

2.1 PreProcessor

The PreProcessor generates a structured prompt by combining a *Selector* and a *Builder*. The *Selector* collects the *MUT Input*, the *MUT*, and the *MUT Tests*. The *Builder* builds the prompt by instantiating a template on the three items and the queries.

Selector. The *Selector* mines three essential items from the input execution scenario: the ***M**ethod **U**nder **T**est MUT* (less than 3K tokens, to comply with the LLMs we currently use in our experiments), the *MUT Tests* (less than 4K tokens), and the *MUT Input* (less than 1K tokens). Selecting the items with upper bounds is important to both match the maximum context length of the LLM and optimize the cost of the query that depends on the number of tokens [57]. Figure 2 shows the *MUT*, *MUT Tests* and *MUT Input* that the *Selector* produces for scenario $\mathcal{S}$.

The *MUT* is the code snippet relevant for understanding the focal method M: the signature and body of M, and the strongly related methods that M directly invokes and that belong to the same Java class of M. *Selector* retrieves such code snippet with JavaParser [55] to statically collect Java code within the method call dependency graph. *Selector* selects the *MUT Tests* as the set of test cases that directly or indirectly (within 3 nested calls) execute M from the test suite of the class of M. We truncate the characters of the test suite to 4K tokens to fit LLM's context limit. *Selector* collects the *MUT Input*, that is, the parameters and the surrounding context code of the call to M, from the Java bytecode with ASM[5], a popular framework for instrumenting bytecode [35]. The Java instrumentation is not the only possible strategy to collect *MUT Input*, but it is also possible to integrate with different carving techniques [21]. *Selector* compares the signature of M with the instrumented methods to retrieve the direct method call, encoded as a string. It extends the method call string with the surrounding context, that is, the code within the ten lines both before and after the method call.

The information about the context of the method is extremely useful for the LLM to understand the invocation. For example, *Selector* builds the *MUT Input* of method `solve` in Defects4J, by augmenting the method call `solver.solve(f, Math.PI, 4)` with the statements `UnivariateRealFunction f = new SinFunction()` and `UnivariateRealSolver solver = new BrentSolver()`, the code snippets that *Selector* extracts from the surrounding context code of `solve`, and that build the actual parameters of the call.

Builder. The *Builder* instantiates the five sections of the *Template* by augmenting *MUT*, *MUT Tests* and *MUT Input* that the *Selector* produces with *Task* and *Queries*. The five sections in Figure 2 shows the *Template* that the *Builder* produces for scenario $\mathcal{S}$:

- **MUT:** The source code of the method `asNode(String node)`, which is the method under test M.
- **MUT Tests:** The test suite of `asNode`, that *Selector* extracts from the suite of Class `PropertiesRedisConnectionDetails`, that includes `asNode`.
- **MUT Input:** The method call of `asNode (String node)` with the input "`[::1]:6379`" that we ask E-Test to classify. The simple call in the example does not require context information.
- **Task:** The instruction about answering five queries for the LLM: Given *MUT*, *MUT Tests* and *MUT Input*, we ask LLM to answer the questions in *Queries*. For each question, we ask LLM for a binary Yes or No answer in JSON format.
- **Queries:** The five queries that we ask LLM to answer to assess *MUT Input* and that we discuss in the next section.

Queries. We assume that LLMs can effectively infer the characteristics of scenarios, thanks to the huge corpus of public bug reports, issues, code, and tests that belong to the training data of the LLM. We formulate the queries about the input scenario, by considering the aspects of a scenario that are likely related to the execution space that the suite tests: the similarities of the scenario with respect to the test cases in the suite, the differences of the execution of the scenario with respect to the behaviors that the suite exercises, the likelihood of a correct result or failure.

[5]https://asm.ow2.io/

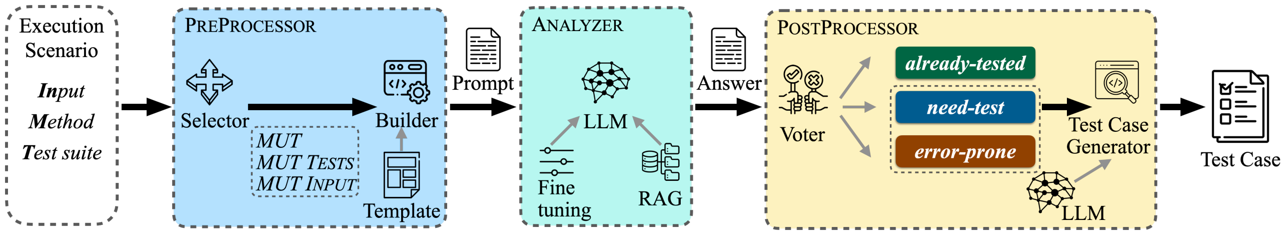


Figure 1: Overview of E-Test

```
MUT:
private Node asNode(String node) {
    String[] components = node.split(":");
    return new Node(components[0],
        Integer.parseInt(components[1]));
}
MUT TESTS:
asNode("127.0.0.1:1111"); asNode("127.0.0.2:2222");
asNode("127.0.0.3:3333"); // without JUnit boilerplate
MUT INPUT:
asNode("[::1]:6379");
TASK:
Answer all questions in QUERIES. For each question, you should
    answer only YES or NO. Return results in a JSON dictionary.
QUERIES:
Q1, Q2, Q3, Q4, Q5 // five queries
```

Figure 2: A structured prompt for the example scenario $\mathcal{S}$ instantiated by E-Test

We formulate five queries about different although partially overlapping characteristics.

Q1: Is *MUT Input* a similar scenario compared with *MUT Tests*? This query concerns the *logical similarity* of the *MUT Input* with respect to the scenarios that the suite tests. Thus it aims to distinguish scenarios that are already tested from scenarios that deserve to be further tested.

Q2: Does *MUT Input* cover more lines or branches than *MUT Tests*? This query complements Q1 by highlighting the differences in code coverage. The extent to which the *MUT Input* exercises new code elements with respect to the *MUT Tests* heuristically identifies areas of the code that the suite does not test yet.

Q3: Will *MUT* work differently when executed under *MUT Input*? This query looks for *anomalous behaviors* that the *MUT Input* reveals. Anomalies in the execution are key elements that deserve particular attention when testing a software.

Q4: Does *MUT* still produce correct results when executed under *MUT Input*? This query estimates the correctness of the *result* of executing the *MUT* with the *MUT Input*, and spotlights scenarios that deserve further testing.

Q5: Will *MUT Input* reveal any bug in *MUT*? This query reinforces Q4, by stressing the possibility of a failure, by asking LLMs to focus on potential bugs in the code.

We formulate the queries with heterogeneous answers with respect to the classification task (the relevant answer for characterizing the senario as relevant for testing may be positive or negative, depending on the question), to reduce the risk of LLMs following a fixed answering pattern, with undesirable biases in the answers. By mixing the answers that identify each category, we encourage LLMs to evaluate each query on its own merits. As a result, the expected answers for classifying a specific scenario are not all Yes or all No, but a combination of Yes or No.

We define questions that LLM can properly understand and answer, by looking for terms that might have been extensively used in the training set of the LLM in many different contexts, and thus with a meaning that LLM can easily interpret. To this end, we identified the most suited terms for the queries about software bugs from Stack Overflow[6], by assuming a key role of Stack Overflow in the training of popular LLMs. We sorted the queries according to the upvote. We selected all terms that occur in the top 10% queries, filtered out the stop words, sorted the terms by frequency, and selected the top terms related to the key terms in the queries. We obtained *similar*, *cover*, *line*, *branch*, *differently*, *correct*, and *reveal*. We use these terms to build the queries.

2.2 Analyzer

The Analyzer queries an LLM to answer the five queries of the prompt. The answers to the five queries for the prompt corresponding to the scenario $\mathcal{S}$ in JSON format are {Q1: NO, Q2: YES, Q3: YES, Q4: NO, Q5: YES}. The input is not similar to any test case in the test suite (Q1: NO), it likely increases the coverage of the test suite (Q2: YES), it likely works differently from any test case in the test suite (Q3: YES), it likely does not produce a correct result (Q4: NO), and it likely reveals a bug in `asNode` (Q5: YES). E-Test implements LLMs with fine-tuning [60], and Retrieval-Augmented Generation (RAG) [34], as illustrated in Figure 1. Here we present the dataset used to validate E-Test and detail the techniques implemented within it.

Dataset. Our dataset consists of 1,975 execution scenarios that we obtained by augmenting an initial set of 1,825 scenarios from Defects4J with 150 scenarios that we mined from GitHub. The generation of the dataset requires about 180 person-hours, to build a dataset reusable in further studies.

Defects4J is a popular benchmark with all the information required to understand, test and replicate software bugs that have been reported from the field, after software has been publicly released [30]. We augmented the scenarios that we obtained from Defects4J with a set of scenarios that we collected by sampling GitHub, to increase the diversity of the dataset with issues about additional libraries and projects.

We mined the closed GitHub issues from highly-starred open-source Java projects (Spring Boot, Apache ShardingSphere, Apache

[6]https://stackoverflow.com/

Table 1: The GitHub dataset used in the experiments

Project	Version	LOC	#Classes	#Test Suites	#Test Cases	Avg. Test Cases / Suite	Avg. LOC / Suite
Spring Boot	3.1.12	683 988	6 901	2 261	13 328	5	135
Apache ShardingSphere	5.5.0	1 519 076	7 635	1 382	5 867	4	92
Apache Dolphinscheduler	3.2.1	257 915	2 336	469	1 988	4	143
Micrometer	1.11.12	166 565	1 231	385	1 684	4	191

Dolphinscheduler and Micrometer [7]). Table 1 shows the statistics of the GitHub dataset that we used in the experiments (version, LOC, number of classes, number of test suites, number of test cases, and averages). We filtered the results by label (*bug* or *defect*), creation date (*after 1 January 2020*), and number of comments (*at least 5*). We carefully selected 50 non-trivial and reproducible bugs that were triggered due to not-yet-tested inputs.

We created a set of 719 *error-prone* scenarios $\langle In, M, T\rangle$ by reproducing 669 Defects4J bugs[8] using the available tests and the 50 bugs from GitHub. We built the *need-test* scenarios from the *error-prone* scenarios by (i) replacing the buggy method M with the repaired method, (ii) reproducing the same execution scenario, and (iii) checking that the test passes. The scenarios that we obtain in this way are missing execution scenarios by construction, since the original fault escapes the testing. The scenarios are not *error-prone* since we build them with the repaired method.

We completed the dataset with 537 *already-tested* scenarios that we obtained by (1) substituting the input In with a new test input that (i) we generated with Evosuite [19], (ii) is not in the test suite T, (iii) succeeds, and (iv) does not increase the branch coverage of T, and (2) keeping the buggy M. The scenarios that we obtain in this way do not contribute to improving the test suite.

The final dataset contains 719 *error-prone* scenarios, 719 *need-test* scenarios, and 537 *already-tested* scenarios. The dataset is available in the replication package [9].

Fine-tuning. Vanilla LLMs often do not perform well for specific tasks, due to their general-purpose nature and training [59]. Chow et al. [13] observe that pretrained LLMs do not usually work best on code-related tasks, and show that the performance of LLMs improves when the LLMs are fine-tuned. The requirements of software testing demand a level of precision and contextual understanding that general LLMs may not provide [57]. Fine-tuning the LLM can improve its capability to accurately classify execution scenarios.

We fine-tuned the LLM with samples from all the three types of scenarios: *already-tested*, *need-test*, and *error-prone*. Each sample consists of a tuple of $\langle$*Prompt*, *Responses*$\rangle$, where *Prompt* is the generated prompt for a scenario $\langle In, M, T\rangle$, and *Responses* is the ground-truth answers to the five queries in JSON format.

We fine-tuned only GPT-3.5 Turbo, since it is one of the best vanilla LLMs for predicting not-yet-tested scenarios. We fined-tuned GPT-3.5 Turbo with samples from all 17 projects of Defects4J. We split the Defects4J part of our dataset into *fine-tuning* and *validation* sets with a 5:95 ratio, and with balanced sets of scenarios of the three types within the *fine-tuning* set. We fine-tuned the model with batch size 1, learning rate multiplier 2.0 and 3 epochs using the balanced fine-tuning dataset.

RAG Retrieval-Augmented Generation. The size limitations of the prompts of vanilla LLMs do not allow to feed the whole set of information in the presence of large code and test suites. We implemented RAG to query the LLM with the whole code base, to overcome the size limitations of the prompts. We built RAG indices for both the source code and the tests with LlamaIndex[10], to feed LLMs with complete information about the software project. RAG extends the *MUT*, *MUT Tests*, and *MUT Input* components of the prompt with code embedding that is strongly related to the prompt in terms of cosine similarity. We embedded code vectors for RAG querying with the OpenAI embedding model `text-embedding-ada-002`.

Few-shot Learning. Vanilla LLMs may not be very precise when fed with plain prompts only. We augmented the LLM with few-shot learning [59], by adding fixed *already-tested*, *need-test* and *error-prone* scenarios that we suitably label, to the head of the prompt, aiming to improve the knowledge we feed to the LLM. We experimented with 3-shot, 6-shot and 9-shot learning, that is, with 1,2 and 3 scenarios per type, respectively. The Analyzer queries the LLM with the fixed scenarios and the corresponding ground truth answers from the Defects4J dataset.

2.3 PostProcessor

The PostProcessor classifies the scenario as *already-tested*, *need-test*, or *error-prone*, and generates test cases. The *Voter* classifies the scenarios based on the highest number of correct answers among the five queries, as shown in Table 2. The *Voter* classifies a scenario as *already-tested* if it is similar to the already-tested scenarios (Q1 = Yes), does not increase the code coverage (Q2 = No), does not work differently from the test cases in the suite (Q3 = No), produces a correct output (Q4 = Yes), and does not expose a bug (Q5 = No), according to the LLM. The *Voter* classifies a scenario as *need-test* if it differs from all the already tested scenario (Q1 = No), increases code coverage (Q2 = Yes), does not work differently form the test cases in the suite (Q3 = No), produces a correct output (Q4 = Yes), and does not expose a bug (Q5 = No), according to the LLM. The *Voter* classifies a scenario as *error-prone* if it differs from all the already tested scenario (Q1 = No), increases code coverage (Q2 = Yes), works differently form the test cases in the suite (Q3 = Yes), produces a wrong output (Q4 = No), and exposes a bug (Q5 = Yes), according to the LLM.

The *Voter* classifies a scenario according to the highest number of answers that match the values in Table 2. The mismatch of the answer with respect to the expected values in Table 2 may depend

[7]Spring Boot, ShardingSphere, Dolphinscheduler, and Micrometer on GitHub
[8]We discarded 166 bugs where *MUT* or *MUT Input* exceeded the context limit.
[9]https://github.com/ketaiq/E-Test-package
[10]https://www.llamaindex.ai/

Table 2: Truth table for each query and scenario

Scenario	Q1	Q2	Q3	Q4	Q5
Already-tested	Yes	No	No	Yes	No
Need-test	No	Yes	No	Yes	No
Error-prone	No	Yes	Yes	No	Yes

on the inaccuracy of the prediction in the answers and the fading boundary between executions and imperfect matching, for instance, a *need-test* scenario may derive from a test that is dissimilar to the existing ones even if it is not likely to cover new code. The *Voter* solves ties with the priority *error-prone* > *need-test* > *already-tested*, to ensure not to miss scenarios that cannot be clearly classified, thus privileging the robustness over the size of the suite.

The *Test Case Generator* generates JUnit test cases for the *MUT* from the identified not-yet-tested scenarios, by continuing the conversation with the LLM.

For example, the *Voter* correctly classifies the scenario $\mathcal{S}$ as *error-prone*, since the Analyzer answers ⟨NO, YES, YES, NO, YES⟩. E-Test augments the original test suite $\mathcal{T}$ with a new test case that reveals the bug.

3 Experimental Validation

Research Questions

We experimentally evaluate E-Test to answer the following research questions (RQs):

- **RQ1 Impact of LLMs: What impact does the choice of the LLM have on E-Test?** We compare the impact of different pre-trained LLMs on E-Test, to measure the contribution of E-Test over vanilla LLMs.
- **RQ2 Comparative Evaluation: Is E-Test better than state-of-the-art approaches?** We compare E-Test with state-of-the-art regression testing and field testing approaches, the closest approaches to E-Test.
- **RQ3 Impact of Queries: Do the different combinations of queries impact the F1-score?** We evaluate the impact of different combinations of queries (including a single generic query) on the results of E-Test, to identify the most effective set of queries.
- **RQ4 Efficiency: How efficiently does E-Test process scenarios?** We evaluate the efficiency of E-Test in terms of response time and token consumption of the LLM core, and we compare the different LLMs that we experimented with.
- **RQ5 Test Case Generation: Does E-Test generate useful test cases?** We evaluate the effectiveness of the JUnit test cases that E-Test generates.

Experimental Setup

Implementation. We implemented E-Test in Python and Java, with GPT family (3.5Turbo, 4Turbo, 4o), DeepSeek R1 family (1.5B, 7B, 14B, 32B and 70B) and Llama3 family (1B, 3B, 8B, and 70B) [11], and we executed the experiments on an Ubuntu 20.04 cluster with four NVIDIA A100 GPUs, each with 40 GB of VRAM. We prompted GPT-3.5 Turbo, GPT-4 Turbo and GPT4o with the OpenAI APIs, and DeepSeek R1 and Llama3 families with Ollama APIs [12] that we deployed locally. We also implemented RAG querying by integrating LlamaIndex with both OpenAI and Ollama APIs.

In our experiments, we set temperature = 2.0 for E-Test and temperature = 0.75 for the DeepSeek R1 and Llama3 families, because we obtained the best F1-score with these values, as shown in Figure 4. We set top_p = 1 as OpenAI suggests for GPT models,[13] and top_p = 0.9, top_k = 40 for DeepSeek R1 and Llama3 families, as the Ollama documentation suggests.[14] We ran each experiment three times and report the average scores. Three repetitions are widely used in the software engineering literature as a pragmatic compromise between statistical reliability of the results and computational cost [63, 64].

The fine-tuning process of E-Test took 684 seconds (11 minutes) with OpenAI fine-tuning API. The RAG indexing of the complete Java code base took on average 0.24 seconds per 1,000 lines of code when experimented on our dataset. Both fine-tuning and RAG indexing are one time effort only, and both are reasonably efficient.

Metrics. We evaluate the classification in terms of precision, recall, and F1-score for each class of scenarios (*already-tested*, *need-test*, and *error-prone*). We compute precision and recall with respect to the ground truth. We compute the score for a given class (for instance *already-tested*), by considering the other two classes (for instance *need-test* and *error-prone*) as negative samples. We compute the F1-score across all the three predicted classes, that is, the average of the three F1-scores associated with each class. All these metrics have been computed with the scikit-learn library.[15] We also record the inference time and consumed tokens of all examined LLMs.

Table 3 presents the results of our experiments that address *RQ1* and *RQ2*. The Table reports precision (*P*), recall (*R*), and F1-score (*F1*) for each class of scenarios, *already-tested* (columns *Already-tested*), *need-test*, (columns *Need-test*) and *error-prone* (columns *Error-prone*). Columns *Not-yet-tested* groups *need-test*, and *error-prone* scenarios, and presents the average F1-score for the two classes that are relevant for testing (Column *Not-yet-tested/Avg. F1*). The last column (*Total Avg. F1*) reports the average F1-score for the whole set of scenarios.

3.1 RQ1: Impact of LLMs

RQ1 investigates the impact of LLMs on E-Test. The *RQ1* rows (label *RQ1* in the first column) report the results of experimenting with vanilla LLMs (label *Vanilla* in the second column), few-shot learning (label *Few-shot learning*) and RAG (label *RAG*). The vanilla experiments classify the input scenarios by instantiating the LLM of Analyzer with GPT family (3.5Turbo, 4Turbo, 4o), DeepSeek R1 family (1.5B, 7B, 14B, 32B and 70B) and Llama3 family (1B, 3B, 8B, and 70B). The results are reported in the rows with the corresponding labels in the third column of the table. The experiments with GPT-4 Turbo augmented with combinations of 3, 6 and 9 shots as the LLM of the Analyzer are reported in the rows labeled *Few-shot*

[11] https://openai.com/, https://www.deepseek.com/ and https://www.llama.com/

[12] https://github.com/ollama/ollama

[13] https://platform.openai.com/docs/api-reference/responses/create

[14] https://github.com/ollama/docs/api.md

[15] https://scikit-learn.org/

Table 3: Precision (P), Recall (R), F1-score (F1) for different models and configurations

		Approach	*Already-tested*			*Not-yet-tested*							Total Avg.
						Need-test			*Error-prone*			Avg.	
			P	R	F1	P	R	F1	P	R	F1	F1	F1
RQ1	**Vanilla**	Llama3 1B	0.29	0.80	0.43	0.39	0.11	0.17	0.40	0.16	0.22	0.19	0.27
		Llama3 3B	0.20	0.00	0.00	0.36	0.33	0.34	0.37	**0.68**	0.48	0.41	0.28
		Llama3 8B	0.26	0.19	0.22	0.36	0.51	0.42	0.36	0.27	0.31	0.37	0.32
		Llama3 70B	0.18	0.29	0.22	0.34	0.28	0.31	0.28	0.19	0.23	0.27	0.25
		Deepseek R1 1.5B	0.27	0.49	0.35	0.35	0.19	0.24	0.38	0.33	0.35	0.30	0.31
		Deepseek R1 7B	0.18	0.12	0.14	0.36	0.44	0.39	0.36	0.40	0.38	0.38	0.30
		Deepseek R1 14B	0.31	0.51	0.37	0.41	0.21	0.27	0.38	0.38	0.38	0.32	0.34
		Deepseek R1 32B	0.21	0.22	0.20	0.36	0.40	0.38	0.34	0.30	0.31	0.34	0.30
		Deepseek R1 70B	0.24	0.56	0.33	0.35	0.10	0.14	0.34	0.27	0.28	0.21	0.25
		GPT-3.5 Turbo	0.26	0.10	0.15	0.35	0.64	0.46	0.36	0.23	0.28	0.37	0.30
		GPT-4 Turbo	0.17	0.20	0.18	0.34	0.56	0.42	0.31	0.07	0.11	0.26	0.24
		GPT-4o	0.19	0.33	0.24	0.32	0.38	0.16	0.37	0.10	0.35	0.25	0.25
	Few-shot learning	GPT-4 Turbo (3-shot)	0.19	0.24	0.21	0.35	0.52	0.42	0.32	0.09	0.14	0.28	0.26
		GPT-4 Turbo (6-shot)	0.15	0.19	0.17	0.35	0.55	0.43	0.32	0.08	0.13	0.28	0.24
		GPT-4 Turbo (9-shot)	0.17	0.24	0.20	0.34	0.54	0.42	0.30	0.04	0.07	0.24	0.23
	RAG	Llama3 $8B_{RAG}$	0.23	0.28	0.26	0.43	0.51	0.47	0.53	0.37	0.44	0.41	0.39
		Llama3 $70B_{RAG}$	0.22	0.51	0.30	0.34	0.22	0.27	**0.62**	0.32	0.42	0.35	0.33
		GPT-3.5 $Turbo_{RAG}$	0.19	0.08	0.11	0.42	**0.71**	0.53	0.60	0.40	0.48	0.51	0.37
		GPT-4 $Turbo_{RAG}$	0.15	0.34	0.21	0.31	0.33	0.32	0.55	0.11	0.19	0.26	0.24
		Deepseek R1 $70B_{RAG}$	0.18	0.29	0.22	0.29	0.24	0.26	0.52	0.45	0.48	0.37	0.32
	Baseline	Random Classifier	0.36	0.33	0.35	0.36	0.33	0.34	0.28	0.33	0.30	0.33	0.33
RQ2	**State-of-the-art**	Cruciani et al. [14]	0.32	0.32	0.32	0.27	0.27	0.27	0.44	0.44	0.44	0.35	0.34
		Gazzola et al. [22]	0.50	**1.00**	0.67	0.00	0.00	0.00	0.18	0.33	0.23	0.12	0.30
	Fine-tuning	E-Test	**0.67**	0.94	**0.78**	0.49	0.26	0.34	0.49	0.58	**0.53**	0.43	**0.55**
		E-$Test_{RAG}$	0.47	0.41	0.44	**0.51**	0.61	**0.56**	0.52	0.46	0.49	**0.53**	0.50

learning in the second column. The experiments of the LLM with RAG are reported in the *RAG* rows of Table 3. The last row of the *RQ1* section of the table (label *Baseline* in the second column) reports the results we obtained with a random classifier (i.e., each type of scenarios has equal probability), which we implemented with `scikit-learn` library that we executed ten independent times on the same dataset.

The results indicate that the vanilla LLMs do not significantly improve over the random classifier. Only two LLMs (Llama3 1B and Deepseek R1 14B) perform slightly better than the random classifier for *already-tested* scenarios. Some LLMs perform better than the random classifier for the *Not-yet-tested* scenarios, but the total average F1-score of the LLMs is consistently lower than the baseline except for Deepseek R1 14B (+1%). Overall, Llama3 8B and GPT-3.5 Turbo perform equally well in identifying not-yet-tested scenarios.

The *Few-shot learning* rows of the table report the results of augmenting GPT-4 Turbo with few-shot learning, which does not improve the results, and we did not invest further effort on few-shot learning.

The *RAG* rows of the table report the results of experimenting with RAG. The F1-scores indicate that RAG only marginally improves the results of the LLMs. RAG obtains the best results with Llama3 8B with an F1-score that improves from 32% to 39%. We

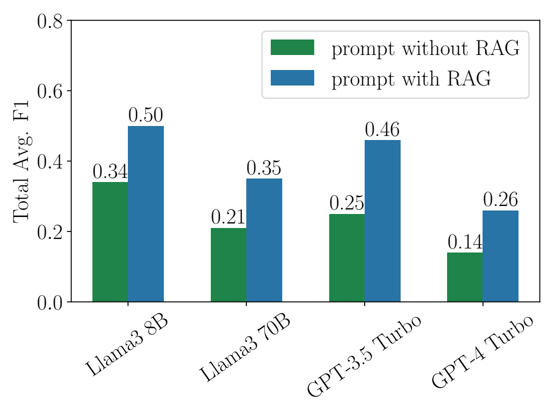
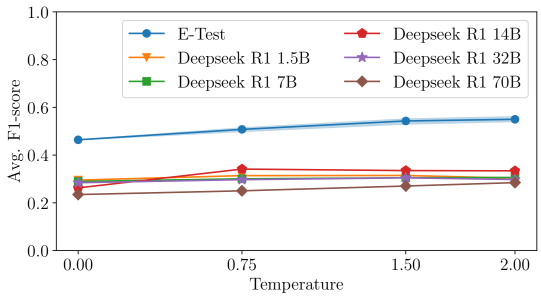
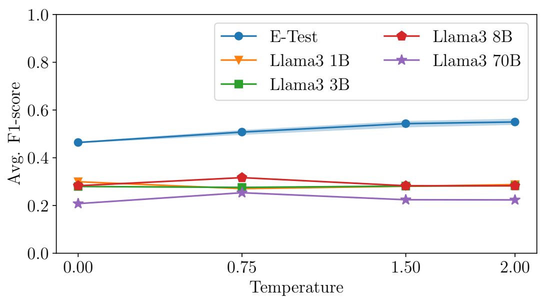


Figure 3: Total Avg. F1-score on large test suite for the LLMs

terminated the experiments with GPT-4 Turbo, since the experiments with RAG are extremely expensive and we decided to stop experimenting, once we had evidence of the relatively small impact of RAG. The test suites of the scenarios in our dataset contain 68 test cases on average, and only 6.6% scenarios of our dataset include test suites with more than 16K tokens, which largely exceed the context window of LLMs, and thus benefit from RAG. Therefore,



(a) Deepseek family



(b) Llama3 family

Figure 4: Inverse-scaling behaviour on E-Test

the results with RAG may underestimate the contribution of RAG. Figure 3 compares the F1-scores of various LLMs prompted with and without RAG, computed only on scenarios with a large test suite that is larger than 16K tokens. The diagrams in the figure show the relevance of RAG to scale up. The overall F1-scores gain at least a 12% improvement with RAG.

The last two rows of the table (label *Fine-tuning* in the second column) report the results of experimenting with E-Test that fine tunes GPT-3.5 Turbo, one of the LLMs that performs the best according to our experimental evaluation. The E-Test fine-tuning of GPT-3.5 Turbo largely improves the total average F1-score, with an overall 80% relative improvement, and significant gains for *already-tested* (+63%) and *error-prone* (+25%) scenarios. We argue that the relative drop of the F1-score for *need-test* scenarios is an acceptable consequence of the improved alignments of E-Test for the scenarios of the other types. E-Test largely improves over the random classifier as well, with a 66.6% relative improvement of the overall F1-score.

The results of E-Test with and without RAG (rows E-Test$_{RAG}$ and E-Test, respectively) confirm the relatively small impact of RAG on the LLMs in this context. E-Test$_{RAG}$ works best for *Not-yet-tested* scenarios (Avg. F1 53%) and specifically for *need-test* scenarios (F1 56%), while E-Test works best for both *already-tested* and *error-prone* scenarios (F1 78% and 53%, respectively).

The large improvement of E-Test for *already-tested* scenarios (78% F1-score, more than double than both the random classifier and all LLMs) paired with the best F1-score for *error-prone* scenarios indicate that E-Test can indeed largely improve the test suite without a combinatorial explosion of the suite.

We ran an inverse-scaling experiment on two families of models—Deepseek R1 (1.5B, 7B, 14B, 32B and 70B) and Llama3 (1B, 3B, 8B and 70B)—using different temperatures (0.0, 0.75, 1.50 and 2.00), with top_p = 0.9 and top_k = 40. Figure 4 shows the average F1-score of each model with various temperatures. E-Test outperforms all LLM models with the best F1-score of 0.55 ± 0.01 over 3 repetitions with temperature = 2. E-Test also maintained an average F1-score of 0.52 ± 0.03 across 4 various temperatures. The slight variance of the average F1-score of E-Test (the tiny shadow around the diagram in Figure 4) shows the stable performance of E-Test.

The experiments indicate that *smaller* LLMs outperform large models (70B) for all temperatures. The Deepseek R1 14B model achieves the best F1-score of 0.34 ± 0.01 with temperature = 0.75 higher than the best F1-score of 0.28 ± 0.05 with temperature = 2 of the 70B model. Similarly, the Llama3 8B model achieves the best F1-score of 0.32 ± 0.00 with temperature = 0.75 higher than the best F1-score of 0.25 ± 0.00 with temperature = 2 of the 70B model. We argue that the results may depend on the frequencies of the types of test cases in the training set, the phenomenon that McKenzie et al. indicate as "unwanted imitation"[40]: there may be a higher percentage of *already-tested* than *not-yet-tested* in large training sets, and this may impact the performance of the models.

RQ1 Findings: The LLM does have an impact on E-Test. E-Test (fine-tuned GPT-3.5 Turbo) outperforms the random classifier baseline, and performs best among the LLMs we evaluated for this task, with potential improvements with RAG for large test suites.

3.2 RQ2: Comparative Evaluation

RQ2 investigates the improvement of E-Test over the state of the art. We compare E-Test against techniques for improving the quality of a test suite: *FAST++* and *field-ready testing*.

FAST++ reduces the size of a regression test suite without executing the test cases: It converts the test cases of the suite to vectors with Term Frequency-Inverse Document Frequency (TF-IDF), clusters the vectors with K-means, and reduces the suites to the test cases in the K centers of the clusters. We classified all the scenarios $\langle In, M, T\rangle$ in our dataset, as *already-tested*, *need-test* and *error-prone* using the inverse order of the prioritization of *FAST++*. Row Cruciani et al. [14] in block *RQ2* of the table reports the results. E-Test outperforms *FAST++* for all indexes, but for the recall of *need-test*, where *FAST++* performs only slightly better (57% over 54%). The large gap of the Total Avg. F1 (50% and 54% for E-Test$_{RAG}$ and E-Test over 34% for *FAST++*) confirms the improvement of E-Test over *FAST++*.

The *field-ready testing* approach of Gazzola et al. [22] generates test cases from scenarios that emerge in production. It selects the scenarios that deserve further testing with a grammar-based mechanism, and instantiates parametric test case templates for the selected scenarios. We experimented with the trigger that [22] report as best performing (1.4%). We classified the scenarios that *field-ready testing* selects from the Defects4J dataset to generate test cases as *error-prone*, and the scenarios that *field-ready testing* does not select as *already-tested*. The *field-ready testing* approach of Gazzola et al. does not allow distinguishing *need-test* from *error-prone* scenarios.

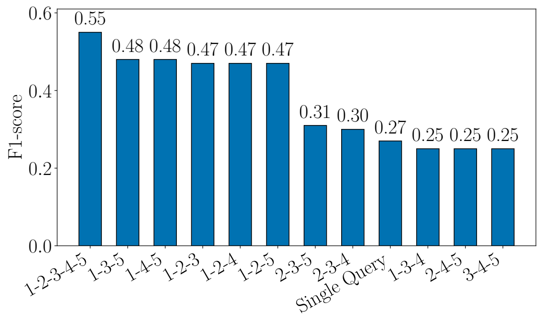


Figure 5: Impact of Query combinations on F1-score (1-2-3 means a combination of Q1, Q2 and Q3 in order)

E-Test outperforms the *field-ready testing* approach for all indexes, but for the recall of *need-test*, where *field-ready testing* does not miss any *already-tested* scenarios, due to the very low optimal trigger that [22] choses to minimize the probability of missing interesting test cases. The 94% result of E-Test is only slightly worse. The large gap of the Total Avg. F1 (50% and 54% for E-Test$_{RAG}$ and E-Test over 30% for *field-ready testing*) confirms the improvement of E-Test over *field-ready testing*.

The F1-score of 0.55 that we obtain for a three-way classification problem is indeed good, since the main literature agrees on considering a F1-score above 0.5 sufficient for less demanding binary classification problems [8, 26, 45]. The comparison to the random baseline (F1-score 0.33), the best rule-based approach (F1-score 0.30), the best learning-based approach (F1-score 0.34), and the best LLM approach (F1-score 0.39) further confirms the quality of the 0.55 F1-score of E-Test.

RQ2 Findings: E-Test performs significantly better than *FAST++* (+61.8% relative improvement) and *field-ready testing* (+83.3% relative improvement), which are the approaches functionally closest to E-Test, in terms of total average F1-score.

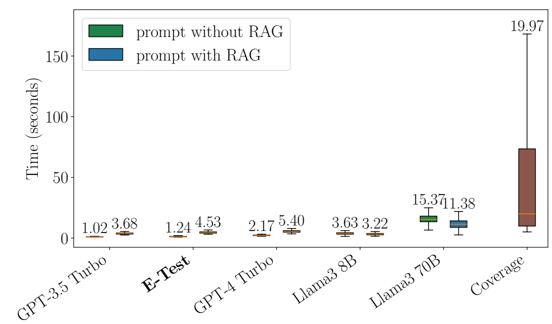
3.3 RQ3: Impact of Queries

RQ3 investigates the impact of the combinations of queries in the prompt of E-Test. Figure 5 shows the F1-scores for all odd combinations of queries (the label of the x-axis), sorted by F1-score. The label *1-2-3-4-5* indicates the five queries of E-Test, the label *Single Query* indicates the single query *"Given* MUT, MUT Input *and* MUT Tests, *you should answer a number indicating its priority for testing. Answer 1 if* MUT Input *is already-tested. Answer 2 if* MUT Input *is need-test. Answer 3 if* MUT Input *is error-prone. Do NOT explain your answer"*. We experimented with only combinations of three queries, since we need (i) at least two queries for a ternary classification, (ii) an odd number of queries to break ties, and (iii) a limited number of queries to keep the size of the prompt within a reasonable limit. We added the single query to study the need of multiple queries.

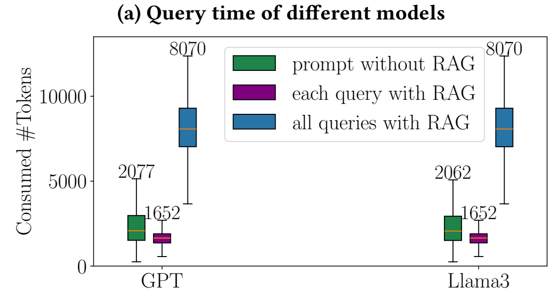
The experiment confirms the better performance of five queries over subsets of three queries. It indicates a nonevent performance of different combinations of three queries, due to the different dependencies among queries. It indicates that five queries clearly outperform a single query, while three queries only do not always perform better than a single query.

RQ3 Findings: The combinations of queries do significantly impact the classification, with the whole set of five queries performing significantly better than any subset of queries, while keeping the prompt within a reasonable size.

3.4 RQ4: Efficiency



(a) Query time of different models



(b) Token consumption for the queries

Figure 6: Query time and token consumption for different configurations (median numbers are at the top of box plots)

The execution time of LLMs to classify the scenarios is the main factors that limits the practical applicability of E-Test. RQ4 investigates the efficiency of E-Test, by measuring both the response time and the token consumption of the LLMs, the potential bottlenecks for the efficiency of E-Test. We measure the response time as the time that E-Test takes to classify the scenarios with the different LLMs, and we compare the results with the time that E-Test takes to select *not-yet-tested* scenarios by simply executing the scenarios and determining the increment of branch coverage with respect to the test suite. In this way, we quantify the gain of LLMs with respect to the equivalence of the scenarios according to branch coverage. We collected the inference time for each prompt with the OpenAI APIs and Ollama APIs for the examined models.

The plots in Figure 6a clearly show that all LLMs (first 5 pairs of plots) perform dramatically better than the direct computation of branch coverage (the last plot in the figure). The figure indicates similar results for all LLMs, with slightly better results of the GPT family with respect to the Llama family, possibly due to both the different training of the models that we used for the experiments and the runtime setup of the experiments (we ran Llama3 models on our powerful cluster and GPT with OpenAPIs[16]). The data in the figure indicate a high variance of computing branch coverage, with a wide range of values, from 5.12 to 168.14 seconds. On the other hand, the response time of all LLMs remains within small boundaries, within a range between 0.81 and 24.97 seconds, in the worst case. The data with and without RAG in the figure indicate the low impact of RAG on the computation time.

Figure 6b compares the token consumption for prompting with RAG for all five queries as a whole (blue), with RAG separately for each query at once (purple) and without RAG (green). The plots indicate no significant differences between GPT and Llama tokenizers. They also show a large impact of RAG: Prompting queries with RAG (blue and purple[17] plots in the figure) consumes more tokens than prompting without RAG (green plots). The lower variance of prompting with RAG separately for each query at once (purple) than without RAG (green) suggests that the token consumption of suitably prompting with RAG grows less than prompting without RAG, and thus improves the efficiency of E-Test.

Querying LLMs about test coverage is less precise than directly executing the test cases. However, querying LLMs about the possibility that a test case increases the coverage of a large test suite is far less expensive than executing the test cases in the suite in all contexts. Our experiments indicate that querying LLMs is an excellent trade-off between the high and impractical costs of executing very large test suites and the precision of the response.

RQ4 Findings: E-Test classifies scenarios efficiently in terms of response time with respect to the direct computation of branch coverage. The limited variance of consumed tokens further confirms the efficiency of E-Test.

3.5 RQ5: Test Case Generation

RQ5 investigates the ability of E-Test to generate useful test cases. We generated test cases for the 673 bugs of Defects4J with accessible test cases that trigger the failures (trigger tests) and that we use as ground truth: We fed E-Test with the scenarios that correspond to the trigger tests, and the test suites of the methods without the trigger tests. We measured the usefulness of the generated test cases as the percentage of generated test cases that are syntactically correct, compile, reveal failures, and obtain the same or higher branch coverage than the ground truth. We performed syntax check, compilation check, and failure check, with up to five retries for fixing syntax errors, compilation errors, and assertion failures for each sequence, as shown in Algorithm 1.

Figure 7 summarizes the results of the experiment. 667 out of 673 generated tests (99.1%) are syntactically correct (line 6 of Algorithm 1). 23 out of 667 tests required retries to fix some syntax errors, with a mean of 2 retries. 580 out of 667 syntactically correct tests (87% with respect to the syntactically correct tests and 86% with respect to generated tests) successfully compile. 151 out of 580 tests required retries during compilation, with a mean of 1.8 retries.

[16]The two environments offer the similar GPUs, according to the publicly available information from OpenAI [6]

[17]The purple plots in the figure report the tokens consumed for each single query with RAG, which is limited by the context window of LLMs. The consumption for five queries is five times the consumption for a single query reported in the plots.

Algorithm 1 Unit Test Case Generation

Require: execution scenario $\langle$ *MUT Input*, *MUT*, *MUT Tests* $\rangle$
Ensure: unit test case UTC **or** None

1: Execute E-Test scenario classification
2: $UTC \leftarrow$ Continue prompting with "Use *MUT Input* to generate only one test case with an assertion for *MUT* but do not catch exceptions."
3: $MAX\_RETRY \leftarrow 5$
4: $retry_{syntax}, retry_{compile}, retry_{assert} \leftarrow 0$
5: **while** $retry_{syntax}, retry_{compile}, retry_{assert} < MAX\_RETRY$ **do**
6:   Parse UTC with Python library javalang
7:   **if** Syntax error in UTC **then**
8:     $retry_{syntax} \leftarrow retry_{syntax} + 1$
9:     $UTC \leftarrow$ Continue prompting with JavaSyntaxError message
10:   **else**
11:     Compile *MUT Tests* with UTC
12:     **if** Compilation error in UTC **then**
13:       $retry_{compile} \leftarrow retry_{compile} + 1$
14:       $UTC \leftarrow$ Continue prompting with compilation error
15:     **else**
16:       Execute UTC
17:       **if** UTC does not trigger failures **then**
18:         $retry_{assert} \leftarrow retry_{assert} + 1$
19:         $UTC \leftarrow$ Continue prompting to correct assertions
20:       **else**
21:         **return** UTC
22: **return** None

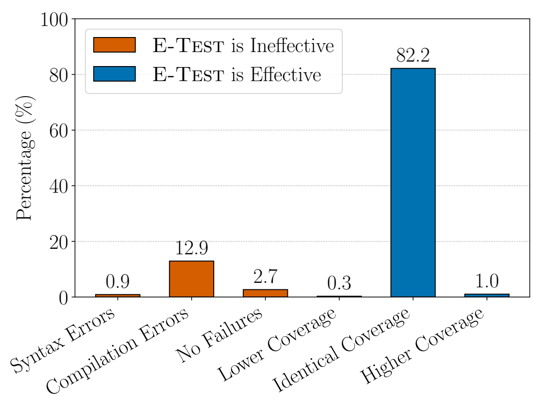


Figure 7: Test generation of E-Test for Defects4J's 673 bugs

562 out of 580 compiled tests (97% with respect to the compiled tests and 83.5% with respect to generated tests) trigger the failure in the ground truth. 85 tests out of 562 required retries to fix assertion issues, with a mean of 1 retry. 560 tests out of 562 tests (99.5% with respect to the failure-revealing tests and 83% with respect to generated tests) reveal the failure with a branch coverage equal or even higher than the ground truth. Overall, E-Test generates test cases that reveal the failures for 83.2% failures in Defects4J.

The results of our experiments indicate that E-Test generates test cases that largely augment the coverage of the code and reveal hidden bugs. The systematic usage of E-Test on the scenarios observed in production can reduce both the gap between the code covered with the test suites and the code executed in production, and the number of bugs that escape the test and occur in production.

RQ5 Findings: E-Test generates a high number of useful test cases, that reveal failures by exercising buggy code.

3.6 Threats to Validity

Internal Validity. The main threat to the internal validity of our results comes from the heterogeneous hardware we use in the experiments (our cluster for Llama and the OpenAI API for GPT.) We mitigate the threat by configuring our cluster with the similar NVIDIA A100 GPUs that OpenAI offers, according to the available documentation. Discrepancies in the hardware configurations can change the results in terms of efficiency (RQ4) but do not impact the validity of E-Test, as an approach that relies on an LLM core.

External Validity. The main threat to the external validity of our results comes from the dataset. Defects4J is a well know dataset that is likely part of the general training set of LLMs. We mitigate the threat by augmenting Defects4J with scenarios we mine from recent Github projects that are less likely used for training LLMs, for their only recent availability on the Web. We also make the whole dataset we use in the experiments publicly available on a replication package to allow interested readers to replicate the experiments.

4 Related Work

E-Test improves the quality of test suites by sifting candidate test cases. Here we briefly discuss test suite augmentation for regression testing and field testing approaches, the closest work that attempts to improve test suites with new test cases and mine new test cases from production, respectively, and we briefly overview the LLM-based approaches to generate test cases that are closest to E-Test.

Test Suite Augmentation. Test suite augmentation approaches generate test cases to reveal changed behaviors between different versions of the program [2, 15, 37, 62, 68]. Santelices et al. [54]'s MaTRIX approach identifies inadequately-tested behaviors with symbolic execution that analyzes the different chains and conditions up to a given distance around the changes, to contain the demands of resources of the technique. Bloem et al. [5]'s approach generates test cases that cover the branches related to functions that change across versions, by combining symbolic execution and model checking. Cruciani et al. [14]'s *FAST++* statically selects representative regression tests from large-size test suites.

E-Test is complementary to test reduction approaches: Test suite reduction approaches optimize test suites, by filtering out redundant test cases, E-Test improves the quality of test suites, by adding test cases that improve the exploration of the runtime space.

Field Testing. Field testing approaches test software systems with scenarios that emerge in production [4]. Most approaches focus on executing tests in production without disrupting the system [10, 42, 52]. Murphy et al. [43]'s in-vivo testing forks methods to execute them with runtime objects that it monitors from production to avoid interfering with the system behavior. Gazzola et al. [22]'s field-ready testing generates test cases from scenarios that emerge in production, by instantiating test templates. E-Test sifts useful test cases from scenarios with an LLM-core, and relies on field testing approaches to both mine scenarios from production and execute the test cases with the configurations from production.

We discuss the experimental comparison of E-Test with both Cruciani et al.'s *FAST++* and Gazzola et al.'s *field-ready testing*, the two closest approaches to E-Test, in Section 3.2.

Large Language Models for Software Testing. Large language models rely on attention mechanism of transformers to answer users' queries[11, 46, 66]. The strong capability of LLMs in understanding code-related tasks provides unique opportunities to analyze software without executing it [44, 48], especially for testing complex software systems [17, 18, 38, 57, 58]. Yang et al. [65] conduct an empirical study about the capability of various open-source LLMs on generating unit test cases and highlight the importance of prompt design and fine-tuning for the task. Kang et al. [31]'s Libro generates bug-reproducing test cases via querying an LLM and augments the test suite with the top test case according to a heuristics-based ranking. Su et al. [56]'s SysKG-UTF generates testing scenarios from bug reports via constructing knowledge graph with LLMs.

We use a fine-tuned LLM to classify execution scenarios using a well-structured prompt template and relevant context with RAG.

5 Conclusion

In this paper, we propose *e'er-improving testing*, and present E-Test. *E'er-improving testing* is a new viewpoint on long lasting testing that upsets the perspective of testing: We can automatically improve test suites by sifting massive amounts of scenarios that become available from many sources, notably the production environment and the automatic generation of large sets of test cases. E-Test fine tunes an LLM core and engineers prompts to gain the information relevant to separate *already-tested*, *need-test* and *error-prone* scenarios of Java methods. It uses the information to identify scenarios that can improve the test suite, namely *need-test* and *error-prone* scenarios, and generates test cases from them. The results of the empirical evaluation show that E-Test can effectively identify not-yet-tested scenarios and generate error-revealing test cases. The comparison with the ground truth and the closest related approaches confirm the improvement of E-Test over the state of the art. In a nutshell, our research contributes to advancing automated software testing methodologies and highlights the potential of LLMs in improving software reliability with an impact for both researchers and developers.

We are currently integrating our approach into system testing pipelines to enhance end-to-end testing efficiency and robustness [7]. We are studying both long context LLMs (context window > 1M tokens) extended with advanced RAG techniques and extensions of the prompt template with queries that can examine scenarios beyond inputs, such as telemetry data (logs, metrics, and traces) to address the core issue of scaling E-Test from unit to integration and system testing, namely the high-quality context.

Acknowledgments

This work is supported by the Swiss SNF project A-Test Autonomic Software Testing (SNF 200021_215487).

References

[1] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference*

on the Foundations of Software Engineering (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 185–196. https://doi.org/10.1145/3663529.3663839

[2] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1572–1584. https://doi.org/10.1109/ICSE43902.2021.00140

[3] Maurício Aniche, Christoph Treude, and Andy Zaidman. 2022. How Developers Engineer Test Cases: An Observational Study. *IEEE Transactions on Software Engineering* 48, 12 (2022), 4925–4946. https://doi.org/10.1109/TSE.2021.3129889

[4] Antonia Bertolino, Pietro Braione, Guglielmo De Angelis, Luca Gazzola, Fitsum Kifetew, Leonardo Mariani, Matteo Orrù, Mauro Pezzè, Roberto Pietrantuono, Stefano Russo, and Paolo Tonella. 2021. A Survey of Field-based Testing Techniques. *ACM Comput. Surv.* 54, 5, Article 92 (May 2021), 39 pages. https://doi.org/10.1145/3447240

[5] Roderick Bloem, Robert Koenighofer, Franz Röck, and Michael Tautschnig. 2014. Automating Test-Suite Augmentation. In *2014 14th International Conference on Quality Software*. 67–72. https://doi.org/10.1109/QSIC.2014.40

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[7] Matteo Brunetto, Giovanni Denaro, Leonardo Mariani, and Mauro Pezzè. 2021. On introducing automatic test case generation in practice: A success story and lessons learned. *Journal of Systems and Software* 176 (2021), 110933. https://doi.org/10.1016/j.jss.2021.110933

[8] Nikolaj Buhl. 2023. F1 Score in Machine Learning. *Erişim adresi: https://encord.com/blog/f1-score-in-machine-learning* (2023).

[9] Ricardo Caldas, Juan Antonio Piñera García, Matei Schiopu, Patrizio Pelliccione, Genaína Rodrigues, and Thorsten Berger. 2024. Runtime Verification and Field-Based Testing for ROS-Based Robotic Systems. *IEEE Trans. Softw. Eng.* 50, 10 (Oct. 2024), 2544–2567. https://doi.org/10.1109/TSE.2024.3444697

[10] Mariano Ceccato, Davide Corradini, Luca Gazzola, Fitsum Meshesha Kifetew, Leonardo Mariani, Matteo Orrù, and Paolo Tonella. 2020. A Framework for In-Vivo Testing of Mobile Applications. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 286–296. https://doi.org/10.1109/ICST46399.2020.00037

[11] Patrick J. Chapman, Cindy Rubio-González, and Aditya V. Thakur. 2024. Interleaving Static Analysis and LLM Prompting. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (Copenhagen, Denmark) *(SOAP 2024)*. Association for Computing Machinery, New York, NY, USA, 9–17. https://doi.org/10.1145/3652588.3663317

[12] Runxiang Cheng, Shuai Wang, Reyhaneh Jabbarvand, and Darko Marinov. 2024. Revisiting Test-Case Prioritization on Long-Running Test Suites. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 615–627. https://doi.org/10.1145/3650212.3680307

[13] Yiu Wai Chow, Luca Di Grazia, and Michael Pradel. 2024. PyTy: Repairing Static Type Errors in Python. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 87, 13 pages. https://doi.org/10.1145/3597503.3639184

[14] Emilio Cruciani, Breno Miranda, Roberto Verdecchia, and Antonia Bertolino. 2019. Scalable Approaches for Test Suite Reduction. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 419–429. https://doi.org/10.1109/ICSE.2019.00055

[15] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. 2019. A snowballing literature study on test amplification. *Journal of Systems and Software* 157 (2019), 110398. https://doi.org/10.1016/j.jss.2019.110398

[16] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically evaluating readily available information for regression test optimization in continuous integration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 491–504. https://doi.org/10.1145/3460319.3464834

[17] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation. *IEEE Trans. Softw. Eng.* 50, 9 (Sept. 2024), 2254–2268. https://doi.org/10.1109/TSE.2024.3428972

[18] Zhiyu Fan, Haifeng Ruan, Sergey Mechtaev, and Abhik Roychoudhury. 2024. Oracle-Guided Program Selection from Large Language Models. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 628–640. https://doi.org/10.1145/3650212.3680308

[19] Gordon Fraser and Andrea Arcuri. 2011. 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* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[20] Gordon Fraser and Andrea Arcuri. 2012. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2012), 276–291.

[21] Alessio Gambi, Hemant Gouni, Daniel Berreiter, Vsevolod Tymofyeyev, and Mattia Fazzini. 2023. Action-Based Test Carving for Android Apps. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 107–116. https://doi.org/10.1109/ICSTW58534.2023.00032

[22] Luca Gazzola, Leonardo Mariani, Matteo Orrú, Mauro Pezzè, and Martin Tappler. 2022. Testing Software in Production Environments with Data from the Field. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 58–69. https://doi.org/10.1109/ICST53961.2022.00017

[23] Luca Gazzola, Leonardo Mariani, Fabrizio Pastore, and Mauro Pezze. 2017. An exploratory study of field failures. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 67–77.

[24] Renan Greca, Breno Miranda, and Antonia Bertolino. 2023. State of practical applicability of regression testing research: A live systematic literature review. *Comput. Surveys* 55, 13s (2023), 1–36.

[25] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. 1993. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.* 2, 3 (jul 1993), 270–285. https://doi.org/10.1145/152388.152391

[26] Maria Cristina Hinojosa Lee, Johan Braet, and Johan Springael. 2024. Performance metrics for multilabel emotion classification: comparing micro, macro, and weighted f1-scores. *Applied Sciences* 14, 21 (2024), 9863.

[27] Joseph R. Horgan, Saul London, and Michael R. Lyu. 1994. Achieving software quality with testing coverage measures. *Computer* 27, 9 (Sept. 1994), 60–69. https://doi.org/10.1109/2.312032

[28] Dong Huang, Qingwen Bu, Yichao Fu, Yuhao Qing, Xiaofei Xie, Junjie Chen, and Heming Cui. 2024. Neuron Sensitivity-Guided Test Case Selection. *ACM Trans. Softw. Eng. Methodol.* 33, 7, Article 188 (Sept. 2024), 32 pages. https://doi.org/10.1145/3672454

[29] Anisha Islam, Nipuni Tharushika Hewage, Abdul Ali Bangash, and Abram Hindle. 2023. Evolution of the Practice of Software Testing in Java Projects. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 367–371. https://doi.org/10.1109/MSR59073.2023.00057

[30] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[31] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-Shot Testers: Exploring LLM-Based General Bug Reproduction. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2312–2323. https://doi.org/10.1109/ICSE48619.2023.00194

[32] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 560–564. https://doi.org/10.1109/SANER.2015.7081877

[33] Zhanyao Lei, Yixiong Chen, Mingyuan Xia, and Zhengwei Qi. 2024. Foliage: Nourishing Evolving Software by Characterizing and Clustering Field Bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1325–1337. https://doi.org/10.1145/3650212.3680363

[34] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 793, 16 pages.

[35] Bolun Li, Pengfei Su, Milind Chabbi, Shuyin Jiao, and Xu Liu. 2023. DJXPerf: Identifying Memory Inefficiencies via Object-Centric Profiling for Java. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO '23)*. Association for Computing Machinery, New York, NY, USA, 81–94. https://doi.org/10.1145/3579990.3580010

[36] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. More Precise Regression Test Selection via Reasoning about Semantics-Modifying Changes. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 664–676. https://doi.org/10.1145/

3597926.3598086

[37] Yujie Liu, Mingxuan Zhu, Jinhao Dong, Junzhe Yu, and Dan Hao. 2024. Compiler Bug Isolation via Enhanced Test Program Mutation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 819–830. https://doi.org/10.1145/3691620.3695074

[38] You Lu, Yifan Tian, Yuyang Bi, Bihuan Chen, and Xin Peng. 2024. DiaVio: LLM-Empowered Diagnosis of Safety Violations in ADS Simulation Testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 376–388. https://doi.org/10.1145/3650212.3652135

[39] Wei Ma, Mike Papadakis, Anestis Tsakmalis, Maxime Cordy, and Yves Le Traon. 2021. Test Selection for Deep Learning Systems. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 13 (jan 2021), 22 pages. https://doi.org/10.1145/3417330

[40] Ian R. McKenzie, Alexander Lyzhov, Michael Martin Pieler, Alicia Parrish, Aaron Mueller, Ameya Prabhu, Euan McLean, Xudong Shen, Joe Cavanagh, Andrew George Gritsevskiy, Derik Kauffman, Aaron T. Kirtland, Zhengping Zhou, Yuhui Zhang, Sicong Huang, Daniel Wurgaft, Max Weiss, Alexis Ross, Gabriel Recchia, Alisa Liu, Jiacheng Liu, Tom Tseng, Tomasz Korbak, Najoung Kim, Samuel R. Bowman, and Ethan Perez. 2023. Inverse Scaling: When Bigger Isn't Better. *Transactions on Machine Learning Research* (2023). https://openreview.net/forum?id=DwgRm72GQF Featured Certification.

[41] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 233–242. https://doi.org/10.1109/ICSE-SEIP.2017.16

[42] Jesús Morán, Antonia Bertolino, Claudio de la Riva, and Javier Tuya. 2017. Towards Ex Vivo Testing of MapReduce Applications. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 73–80. https://doi.org/10.1109/QRS.2017.17

[43] Christian Murphy, Gail Kaiser, Ian Vo, and Matt Chu. 2009. Quality Assurance of Software Applications Using the In Vivo Testing Approach. In *2009 International Conference on Software Testing Verification and Validation*. 111–120. https://doi.org/10.1109/ICST.2009.18

[44] Yu Nong, Haoran Yang, Long Cheng, Honxin Hu, and Haipeng Cai. 2025. APPATCH: Automated Adaptive Prompting Large Language Models for Real-World Software Vulnerability Patching. In *34th USENIX Security Symposium (USENIX Security 25)*. 1–20.

[45] Juri Opitz and Sebastian Burst. 2019. Macro f1 and macro f1. *arXiv preprint arXiv:1911.03347* (2019).

[46] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2025. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Trans. Softw. Eng. Methodol.* 34, 2, Article 42 (Jan. 2025), 28 pages. https://doi.org/10.1145/3697010

[47] Rongqi Pan, Mojtaba Bagherzadeh, Taher A. Ghaleb, and Lionel Briand. 2022. Test case selection and prioritization using machine learning: a systematic literature review. *Empirical Softw. Engg.* 27, 2 (mar 2022), 43 pages. https://doi.org/10.1007/s10664-021-10066-6

[48] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants?. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1144, 25 pages.

[49] Mauro Pezzè and Michal Young. 2008. *Software testing and analysis: process, principles, and techniques.* John Wiley & Sons.

[50] Phuoc Pham, Vu Nguyen, and Tien Nguyen. 2023. A Review of AI-augmented End-to-End Test Automation Tools. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 214, 4 pages. https://doi.org/10.1145/3551349.3563240

[51] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 33, 11 pages. https://doi.org/10.1145/2393596.2393634

[52] Adam Porter, Cemal Yilmaz, Atif M. Memon, Douglas C. Schmidt, and Bala Natarajan. 2007. Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance. *IEEE Transactions on Software Engineering* 33, 8 (2007), 510–525. https://doi.org/10.1109/TSE.2007.70719

[53] Debra J. Richardson and Lori A. Clarke. 1981. A partition analysis method to increase program reliability. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, USA) *(ICSE '81)*. IEEE Press, 244–253.

[54] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2008. Test-Suite Augmentation for Evolving Software. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, USA, 218–227. https://doi.org/10.1109/ASE.2008.32

[55] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. 2023. JavaParser: Visited. https://leanpub.com/javaparservisited Last accessed 22 October 2024.

[56] Yanqi Su, Dianshu Liao, Zhenchang Xing, Qing Huang, Mulong Xie, Qinghua Lu, and Xiwei Xu. 2024. Enhancing Exploratory Testing by Large Language Model and Knowledge Graph. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 98, 12 pages. https://doi.org/10.1145/3597503.3639157

[57] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Trans. Softw. Eng.* 50, 4 (April 2024), 911–936. https://doi.org/10.1109/TSE.2024.3368208

[58] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025. TESTEVAL: Benchmarking Large Language Models for Test Case Generation. arXiv:2406.04531 [cs.SE] https://arxiv.org/abs/2406.04531

[59] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. 2020. Generalizing from a Few Examples: A Survey on Few-shot Learning. *ACM Comput. Surv.* 53, 3, Article 63 (June 2020), 34 pages. https://doi.org/10.1145/3386252

[60] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2025. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models. *ACM Trans. Softw. Eng. Methodol.* (Jan. 2025). https://doi.org/10.1145/3714461 Just Accepted.

[61] Dietmar Winkler, Pirmin Urbanke, and Rudolf Ramler. 2022. What do we know about readability of test code?-a systematic mapping study. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1167–1174.

[62] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. 2010. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 257–266. https://doi.org/10.1145/1882291.1882330

[63] Pengyu Xue, Linhao Wu, Zhen Yang, Chengyi Wang, Xiang Li, Yuxiang Zhang, Jia Li, Ruikai Jin, Yifei Pei, Zhaoyan Shen, et al. 2025. ClassEval-T: Evaluating Large Language Models in Class-Level Code Translation. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1421–1444.

[64] Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. CodeTransOcean: A Comprehensive Multilingual Benchmark for Code Translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 5067–5089. https://doi.org/10.18653/v1/2023.findings-emnlp.337

[65] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 2024. On the Evaluation of Large Language Models in Unit Test Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1607–1619. https://doi.org/10.1145/3691620.3695529

[66] Xueqi Yang, Mariusz Jakubowski, Li Kang, Haojie Yu, and Tim Menzies. 2024. SparseCoder: Advancing source code analysis with sparse attention and learned token pruning. *Empirical Softw. Engg.* 30, 1 (Dec. 2024), 30 pages. https://doi.org/10.1007/s10664-024-10558-1

[67] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120.

[68] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2011. An Empirical Study of JUnit Test-Suite Reduction. In *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE '11)*. IEEE Computer Society, USA, 170–179. https://doi.org/10.1109/ISSRE.2011.26

[69] Zhi Quan Zhou, Shaowen Xiang, and Tsong Yueh Chen. 2016. Metamorphic Testing for Software Quality Assessment: A Study of Search Engines. *IEEE Transactions on Software Engineering* 42, 3 (2016), 264–284. https://doi.org/10.1109/TSE.2015.2478001