How well LLM-based test generation techniques perform with newer LLM versions?

Michael Konstantinou
SnT, University of Luxembourg
michael.konstantinou@uni.lu

Renzo Degiovanni
Luxuembourg Institute of Science and Technology
renzo.degiovanni@list.lu

Mike Papadakis
SnT, University of Luxembourg
michail.papadakis@uni.lu

***Abstract*—The rapid evolution of Large Language Models (LLMs) has strongly impacted software engineering, leading to a growing number of studies on automated unit test generation. However, the standalone use of LLMs without post-processing has proven insufficient, often producing tests that fail to compile or achieve high coverage. Several techniques have been proposed to address these issues, reporting improvements in test compilation and coverage. While important, LLM-based test generation techniques have been evaluated against relatively weak baselines (for todays' standards), i.e., old LLM versions and relatively weak prompts, which may exacerbate the performance contribution of the approaches. In other words, stronger (newer) LLMs may obviate any advantage these techniques bring. We investigate this issue by replicating four state-of-the-art LLM-based test generation tools, HITS, SymPrompt, TestSpark, and CoverUp that include engineering components aimed at guiding the test generation process through compilation and execution feedback, and evaluate their relative effectiveness and efficiency over a plain LLM test generation method. We integrate current LLM versions in all approaches and run an experiment on 393 classes and 3,657 methods. Our results show that the plain LLM approach can outperform previous state-of-the-art approaches in all test effectiveness metrics we used: line coverage (by 17.72%), branch coverage (by 19.80%) and mutation score (by 20.92%), and it does so at a comparable cost (LLM queries). We also observe that the granularity at which the plain LLM is applied has a significant impact on the cost. We therefore propose targeting first the program classes, where test generation is more efficient, and then the uncovered methods to reduce the number of LLM requests. This strategy achieves comparable (slightly higher) effectiveness while requiring about 20% fewer LLM requests.**

***Index Terms*—Unit Test Generation, Large Language Models, Testing and Analysis, AI for SE**

I. Introduction

Recent advances in AI, driven by powerful Large Language Models (LLMs), have revolutionized Software Engineering tasks, including code generation, vulnerability detection, program repair [1], and more recently, automatic test generation [2]–[6]. Despite suboptimal LLM performance (e.g., hallucination), the effectiveness of LLM-based test generation techniques has been significantly improved by combining advanced prompting strategies (e.g., Chain-of-Thought and Self-Consistency) [7], [8] and retrieval augmentation [9] with traditional software engineering techniques, such as code coverage [10] and symbolic execution [11].

All these techniques were effective to mitigate the limitations of LLMs at that time, mainly GPT 3.5. Considering that recent LLMs have improved their reasoning skills, adaptability to various tasks (Mixture-of-Experts) and increased computational efficiency, we raise the question whether the existing LLM-Based test generation techniques remain effective or rather become obsolete, and just by using simple prompting the LLM can produce quality test suites.

To answer this question, we evaluate four LLM-based test generation techniques for Java programs (HITS [5], SymPrompt [11], TestSpark [12] and Coverup [10]), representative of the state-of-the-art in terms of effectiveness and diversity of the underlying techniques. In addition, we include LLM-Plain, a zero-shot simple prompting baseline. All five approaches are executed on the latest available models, namely GPT-4o-mini, Llama 3.3 70B, and DeepSeek V3. We carefully selected 6 Java projects, by considering the training date of the models, and run the five approaches to generate test cases on the entire project (i.e., on every class under test). We compare the performance of the aforementioned approaches on three dimensions.

- We compare the *effectiveness* of the methods in terms of standard test adequacy metrics [13], namely line coverage, branch coverage, and the mutation score [14].
- We compare the methods' *efficiency* (cost) in terms of the number of LLM queries required by each approach to generate the test suites (this represents a real-cost component in the case of the commercial models).
- We study how test effectiveness and involved cost is affected by the targeted *granularity level* at the LLM prompt, i.e., when generating tests for specific methods or for entire classes.

Perhaps surprising, our results show that LLMs can produce relatively effective test suites, outperforming the four state-of-the-art tools. Plain-LLM obtains, in total, 49.95%, 35.33%, and 33.82% on line/branch/mutation coverage, while HITS obtains 42.43%, 29.49%, and 27.97%, SymPrompt 40.95%, 27.19%, and 28.40%, TestSpark 23.65%, 16.48%, and 13.88%, and CoverUP 38.85%, 25.11%, and 26.26%, respectively. This indicates that with new, relatively strong LLMs, test generation methods provide a limited or no advantage to the generation process.

Our results also show that, across the three LLMs used on the experiments, SymPrompt was the least costly tool to run, making 5,602 requests to the LLMs, on average, followed by TestSpark with 7,023 requests, Plain-LLM with 11,815, CoverUP 12,741, and finally HITS, the most costly approach, with 16,735 requests.

While previous approaches were applied at the method level (i.e. the tools generate tests for each method under test), we also evaluate the Plain-LLM prompt at class-level granularity, i.e., the entire class is given as input to the LLM, for which a entire test suite has to be generated. In total, Plain-LLM at class-level granularity obtains 33.53%, 21.05% and 22.08% of line/branch/mutation coverage, but needs only 1,562 requests to the LLMs. Although the generated class-level suites may look less effective than the method-level ones, we observe that these are indeed complementary. When both suites are combined, they reach, in total, 53.67%, 38.74% and 36.55% of line/branch/mutation coverage, and improvement of 7.45%, 9.65%, and 8.07%, respectively, but requiring 13,228 LLM requests.

Although the empirical results are promising and suggest that recent and future releases will keep improving LLMs' effectiveness, there is a clear need to reduce the cost while preserving effectiveness. With this spirit in mind, we explore whether a hybrid approach can be a promising solution for this issue and thus, we adapt the Plain-LLM prompting to first perform the class-level generation, and then the method-level generation, targeting only the methods for which some branches remain uncovered. We observe that a hybrid granularity prompting can generate suites that reach, in total, 52.30%, 38.84% and 36.76% of line/branch/mutation coverage, with only 10,548 LLM requests. Although these hybrid test suites are almost as effective as the combined test suites, we can save 2,680 requests, which constitute almost **20% of cost savings**.

Finally, we observe a key challenge that prevails and needs to be solved by future LLM-based test generation tools. On average, near 20% of the generated tests do not compile, and almost 43% of the tests do not pass (at least one assertion fails). Fixing this great number of non-compiling and non-passing tests can significantly improve their performance, while reducing the costs.

In summary, the key observations of this paper are:

- Current LLMs have certainly improved their reasoning and generation skills, and even with simple prompts can produce test suites as effective as the ones generated with well engineered tools in the state-of-the-art.
- We observe that test suites generated at the class-level granularity can complement the test suites generated at method-level granularity.
- A simple hybrid approach that adapts the granularity-level of the generation (targeting first the classes, and then the uncovered methods) can reduce up to 20% the number of LLM requests, while preserving effectiveness. This is a promising line for future research that we encourage researchers and practitioners to explore.
- Finally, one key challenge that future techniques should aim to address regards the high number of non-compiling and non-passing generated tests that, if fixed, can significantly improve the cost-effectiveness of the LLM-based test generation tools.

II. Background and Related Work

ChatTester is one of the first studies that used LLMs to generate unit tests [4]. ChatTester instructs the LLMs to generate tests for a given class and then iteratively improves them by providing feedback (the compiler error message) every time an error is thrown, i.e., when the test is not compiling. This scheme improves the test suites by generating 34.30% more compiling tests and 18.7% more passing tests. This feedback-driven test repair loop was later adopted by many LLM-based unit test generators [15]–[22]. For instance, ChatUniTest [23] includes a test repair mechanism for simple syntactic errors such as imports and missing semicolons, which are taken from the compilation error message.

TestART [24] is another technique that uses template-based and re-prompting to perform test compilation fixing. It relies on five templates to fix common compilation issues. TestSpark [12] aims to enrich the prompts with some context, i.e., by including the class under test together with the relevant parent classes. CoverUp [10] introduced the idea of providing coverage feedback, i.e., uncovered code, to LLMs so that they can effectively generate tests for uncovered code.

SymPrompt [11] prompts LLMs with information drawn from traditional symbolic execution. SymPrompt statically analyzes a method to identify all feasible execution paths, and then prompts the LLM to generate a corresponding test case for each path. PANTA [25] identifies uncovered code segments and iteratively prompts the LLMs with candidate execution paths as the target.

HITS [5] is a tool that goes deeper into a method before generating tests through slicing. Using the "Chain of Thought" prompting strategy, it instructs the LLM to decompose a given method into smaller, logically coherent code blocks known as slices. It then prompts the model to generate a corresponding test class for each slice.

TELPA [26] computes a call graph to identify methods' sequence calls that are then used as context in the prompt for test generation. There are also some approaches aiming to generate unit tests to *kill mutants* [27] [28]. Notably, Mutap [29], ACH [30], PRIMG [31] are some of the approaches that have been used to generate unit tests to improve the mutation score of generated tests. These approaches differ from the above ones by providing mutation feedback (live mutants) instead of code coverage.

In general, each of the above approaches focuses on different aspects to make the unit test generation process more effective. In this paper, we select HITS [5], SymPrompt [11], TestSpark [12] and Coverup [10] as representative of the aforementioned approaches and evaluate their performance in recent LLMs.

Here it must be noted that the experimental evaluation we follow in this paper has *significant differences* with that of the previous studies. These are:

- **We evaluate the selected methods on SOTA versions of LLMs**. Most of the existing methods were evaluated on GPT 3.5, which was one of the best LLMs at the time the papers were published, while we are using stronger more recent LLMs belonging to different families, namely, gpt-4o-mini, Llama 3.3. 70B, and DeepSeek V3.
- **We evaluate the selected methods on a carefully selected set of projects (entire projects not selected classes)**. Interestingly, the related studies where evaluated on a self picked set of Java classes, which may have unintentionally introduce a selection bias. To strengthen the related analysis/results, we replicate these methods by running them on entire projects instead of selected classes, which will give us a better understanding of the capabilities of the studied tools and approaches.

III. Research Questions

We start by studying the performance of LLM-based test generation techniques on recent LLMs, and thus ask:

A. RQ1: What is the relative effectiveness of LLM-based test generation techniques when using recent LLMs?

To answer to this question we include four state-of-the-art LLM-based test generation tools (HITS [5], SymPrompt [11], TestSpark [12] and Coverup [10]), and one LLM-Plain prompting we implemented to use in our experiments. We measure the effectiveness of the generated test suites using standard test adequacy metrics [13], namely line coverage, branch coverage, and the mutation score [14].

Since the effectiveness of these techniques comes at a cost, we wonder:

B. RQ2: What is the relative efficiency (cost) of these LLM-based test generation techniques?

To answer to this question we measure the number of LLM queries to determine the cost of each workflow's execution (for commercial models, this represents a real-cost component).

Finally, since LLM-plain test generation can be applied at different granularity levels, in particular, at method-level or at class-level, we ask:

C. RQ3: How the granularity level (Class vs Method) affect the effectiveness and cost of the LLM-based test generation techniques?

To answer to this question, we adapt our LLM-Plain strategy to generate test cases for the entire class under test or only a method definition, and thus, we study how this affect on test suites' effectiveness and the number of LLM requests needed.

TABLE I
Projects used

Project	Abbr	JDK	# CUT	# MUT	Previously used	Seen
Binance / Binance connector 2.0.0	BC	8	46	481	ChatUniTest	Yes
bhlangonijr / Chesslib	CH	11	43	538	Gitbug Java	Yes
AuthMe / ConfigMe	CM	8	75	541	Gitbug Java	Yes
AWS / Event-ruler	ER	8	53	656	HITS	No
Flmelody / Windward	W	8	78	400	HITS	No
Apple / Batch-processing-gateway	BPG	17	98	1041	HITS	No
TOTAL			**393**	**3657**		

IV. Experimental design

A. Projects used

For our experiments, we selected six complete open-source Java repositories, representing, to the best of our knowledge, the largest dataset used to date in studies of LLM-based test generation for Java. We selected repositories previously used in LLM-based test generation studies, including those employed for evaluating ChatUniTest [23] and HITS [5], as well as the GitBug-Java dataset [32]. To reduce the risk of data contamination, we deliberately chose repositories such that half are not included in the model's training set. To distinguish between seen and unseen data, we applied the model's announced cutoff date, classifying all projects created after October 1, 2023, as unseen. Table I shows the projects used and the repositories that might be included in the LLM's training set.

Unlike prior studies, which typically evaluate only a subset of classes or methods, our evaluation encompasses all classes within each repository. This comprehensive approach avoids introducing bias through selective sampling and ensures that no approach is inadvertently favored.

B. Metrics used

Effectiveness: We evaluate the effectiveness of the generated test suites using standard test adequacy metrics [13], namely line coverage and branch coverage. Both metrics are measured using JaCoCo, a widely adopted Java code coverage tool that has been extensively used in prior studies on automated test generation. In addition, we computed the mutation score achieved by the generated test suites using PIT [14], a widely used mutation testing framework, with its default configuration.

Efficiency: We use the number of LLM queries to determine the cost of the execution of each workflow, since the LLM queries represent the most computationally intensive part of the approaches studied [4]. For commercial models, queries also represent a real-cost component. Please note that failed API requests caused by unexpected external errors (e.g., network timeouts) are excluded from the analysis. Hence, the cost reported represents the exact number of requests that each implementation under evaluation requires.

Significance: We employ the Mann–Whitney U test [33] to assess whether observed differences between approaches are statistically significant. Following standard practice, we consider results to be statistically significant when the p-value is less than 0.05.

Compilation and passing rates: To compute effectiveness metrics, all failing tests must be excluded. We classify failing tests as either non-compiling or non-passing. Although the evaluated LLM-based tools partially filter invalid tests, we apply a uniform post-processing step for consistency:

- Remove test methods, classes, or helper classes that do not compile.
- Remove files that fail to compile entirely (e.g., due to import errors). Minor naming mismatches are automatically corrected.

Based on such a filtering mechanism, we compute the compilation rate as such:

$$\text{Compilation Rate} = \frac{N_{\text{generated}} - N_{\text{non-compiling}}}{N_{\text{generated}}} \quad (1)$$

For the passing rate, we consider tests that compile and pass. We measure the passing rate as such:

$$\text{Passing Rate} = \frac{N_{\text{generated}} - (N_{\text{non-compiling}} + N_{\text{non-passing}})}{N_{\text{generated}}} \quad (2)$$

C. LLM-based unit test generators

We used recent state-of-the-art LLM-based test generation approaches to evaluate their current effectiveness. The original evaluations of these approaches were conducted using an older such as GPT-3.5-turbo or GPT-4. In our experiments, we ran each approach with its default configuration, but replaced the underlying model with its successor, GPT-4o-mini. Each of the following approaches is built on top of a variation of the LLM-Plain workflow illustrated in Figure 1, and each approach has a unique generation technique. Namely, we employ the following approaches:

HITS [5] uses the LLM to decompose methods into code chunks. Then, it applies the LLM-based unit test generation workflow on each of the code chunk.

SymPrompt [11] attempts to identify execution paths similarly to symbolic execution. Once all execution paths are identified, SymPrompt will use the LLM to generate unit tests for each path.

TestSpark [12] can use either traditional test generation via EvoSuite or LLM-based test generation. When employing LLMs, *TestSpark* prompts the model to generate test cases targeting 100% coverage for the entire class or method under test. Additionally, if the class under test inherits from a parent class, *TestSpark* includes the corresponding parent class code in the prompt. In our experiments, we exclusively used the LLM-based configuration, since we evaluate LLM-based tools previously assessed with earlier model versions.

CoverUp [10] extends prior test generation workflows by introducing an additional coverage augmentation phase. After generating an initial test suite, CoverUp measures code coverage, identifies uncovered lines, and subsequently guides the LLM to generate additional tests targeting the remaining uncovered code.

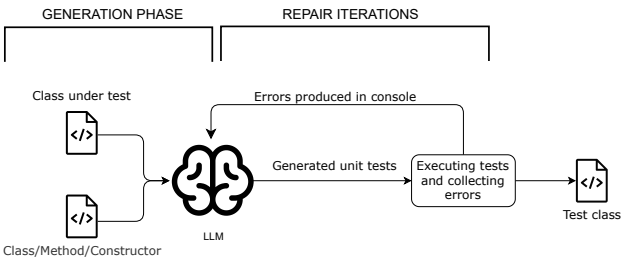

Fig. 1. LLM-Plain unit test generation workflow

Although further approaches have been tried so far, we did not include every approach into our evaluation either because the approach does not contain a publicly available implementation (e.g. ASTER [15]), or simply because they have not been evaluated with older versions of the LLMs of this evaluation study (e.g. PANTA [25]). Additionally, certain approaches, such as *ChatTester*, were excluded despite having been evaluated with earlier versions of our chosen LLMs, as they have been consistently outperformed by one or more of the selected baseline tools.

D. LLM-Plain

LLM-Plain (not to be confused with "Raw LLM") primarily leverages LLM capabilities with minimal parsing. The aim of this implementation is to provide a simplistic approach in which one could build on top of it.

As shown in Figure 1, the approach uses a single generation prompt to produce unit tests for a given class or method, with the full class code as input. Then, a maximum of five repair iterations is applied, just like in the aforementioned baselines. In each iteration, the generated tests are executed, and any errors are sent back to the LLM for correction. The full communication history is maintained throughout, preserving context across iterations. Minimal parsing ensures that the package name, imports, and test class name match the class under test, guaranteeing syntactic consistency. This minimal parsing is similar to the baselines' rule-based fixing. To reduce nondeterminism, the LLM's temperature is set to 0.1.

We consider this approach as "LLM-Plain", as it relies mostly on the capabilities of the LLMs and not on any other existing techniques, static analysis or prompt engineering. Depending on the scope on which we run LLM-Plain, the workflow is slightly changed as follows:

1) Class-level workflow: Given the entire class under test, the LLM is asked to generate tests for the whole class into a single test file (e.g. FooTest will be the only generated test file testing class Foo).

Generation Prompt (Class)

```
The following class is missing unit tests.
Please generate all tests needed to achieve
100% code coverage using Java and Junit5.
Return only the code
<class content>
```

2) Method-level workflow: The workflow is similar to the aforementioned class-level testing, but repeated for each method. Given the entire class under test, LLM-Plain executes the generation process for each method name. As a result, LLM-Plain will not generate a test class for each cut, but a test class for each mut. In addition, if the class under test contains at least one constructor, LLM-Plain will repeat the generation process one more time, asking the LLM to generate tests to cover the constructors of the class.

Generation Prompt (Method)

```
The following class is missing unit tests
for method <method>. Please generate all
tests needed to achieve 100% code coverage
for method <method>, using Java and Junit5.
The name of the generated test class must
be <class name>. Return only the code
<class content>
```

V. Results

A. RQ1: Effectiveness

Table II records the code coverage achieved by all the approaches we study. The approach obtained the highest coverage for each project is highlighted in **bold**, while the second-best performance is underlined. At the bottom of the table, we report the overall coverage across the entire dataset, over all included projects. The effectiveness of Plain-LLM at the method level is reported as Plain (M).

Based on the line coverage results, the Plain-LLM approach consistently outperforms all previously proposed techniques in total and average coverage. Although Plain-LLM does not achieve the highest coverage in every individual repository, its performance remains consistently comparable across projects, resulting in the highest overall effectiveness.

For branch coverage, HITS achieves the highest coverage in the majority of individual repositories; however, when aggregating results across the entire dataset, Plain-LLM covers a larger total number of branches. On average, the differences among Plain-LLM, HITS, and *SymPrompt* are modest. Regarding mutation testing, Plain-LLM again emerges as the most effective approach in terms of total mutation score, with *SymPrompt* ranking second.

Overall, these results indicate that even without sophisticated error parsing or extensive engineering, a Plain-LLM approach that relies primarily on the inherent capabilities of large language models can outperform prior techniques in terms of line coverage and mutation score, while achieving comparable performance in branch coverage.

Statistical tests also confirm this observation. Here, it must be noted that since the effectiveness differences are relatively small, the key claim is that the other methods do not offer any advantage over Plain-LLM.

Finding 1: LLM-based test generation methods have comparable test effectiveness (in fact, Plain-LLM is slightly superior) with Plain-LLM.

B. RQ2: Efficiency

Table III records the number of calls each approach requests to the LLM, that is, the number of API queries required to generate test suites for all repositories in our dataset. Please notet that this number actually represents the most costly part of the approaches. The results show that approaches such as *SymPrompt* and *TestSpark* generally require fewer requests. Notably, the Plain-LLM approach requires fewer LLM queries than both *CoverUp* and *HITS*, while achieving higher test effectiveness.

The higher query cost of *CoverUp* can be attributed to its additional coverage augmentation phase, which triggers additional LLM interactions beyond the initial test generation. In contrast, Plain-LLM does not incorporate such a post-processing step. *HITS*, on the other hand, applies a finer-grained strategy by decomposing methods into smaller code chunks and invoking the test generation workflow for each chunk. Although this enables more targeted test generation, it nearly doubles the number of LLM requests compared to Plain-LLM.

SymPrompt is also applied at the method level, but it goes a step further by focusing on generating tests for individual execution paths. This is an attempt to make the LLM focus on different execution for each method by providing specific and concise targets to cover when producing tests. Interestingly, this results in making fewer calls to generate tests than Plain-LLM; however, such a targeted approach does not necessarily give superior test effectiveness.

Finding 2: Plain-LLM offers a competitive balance between efficiency and effectiveness by making comparable number (fewer) of LLM calls to the other methods. Interestingly, the second most effective approach, HITS, requires nearly twice as many LLM queries as Plain-LLM does.

C. RQ3: Granularity

Table IV records the coverage achieved by Plain-LLM when instructed to generate tests at the class level in contrast to the method level. To further investigate the observed differences between these two configurations, we additionally merged the passing test suites produced by both class-level and method-level prompting (combined approach). We then measured the combined coverage of both sets of generated tests.

TABLE III
Number of Requests to the LLM

Project	HITS	SymPrompt	TestSpark	CoverUp	Plain (Method)
BPG	3985	994	967	2047	2196
ER	2530	1081	1100	1579	2429
BC	9167	2693	1404	3545	2498
W	3216	1381	2189	1916	1350
CH	3703	1493	1597	2222	1859
CL	1264	771	706	955	1386
Total	23865	8413	7963	12264	11718

TABLE II
EFFECTIVENESS BETWEEN HITS, SYMPROMPT, TESTSPARK, COVERUP, PLAIN-CLASS, AND PLAIN-METHOD

Project	Line Coverage						Branch Coverage						Mutation Score					
	HITS	SYMP	TSP	CUP	Plain (C)	Plain (M)	HITS	SYMP	TSP	CUP	Plain (C)	Plain (M)	HITS	SYMP	TSP	CUP	Plain (C)	Plain (M)
BPG	35.52%	27.98%	17.05%	25.20%	36.47%	**45.13%**	**35.89%**	30.41%	19.04%	25.39%	22.88%	32.60%	27.35%	23.76%	14.34%	19.62%	29.33%	**41.21%**
ER	8.96%	13.21%	15.81%	10.66%	35.58%	**45.55%**	5.00%	8.70%	10.13%	7.48%	23.36%	**32.10%**	6.61%	10.56%	11.41%	9.08%	23.03%	**31.53%**
BC	71.47%	83.02%	18.41%	**84.05%**	15.21%	49.91%	**79.45%**	65.75%	36.07%	70.32%	36.20%	49.77%	45.32%	49.30%	10.77%	**50.12%**	7.85%	15.93%
W	45.28%	**48.82%**	17.95%	38.70%	37.14%	36.96%	41.00%	**44.02%**	5.98%	31.08%	24.90%	27.49%	32.35%	**39.60%**	6.84%	27.06%	30.48%	32.81%
CM	62.40%	54.68%	34.61%	54.22%	41.62%	**65.39%**	**51.48%**	44.07%	27.78%	44.81%	25.56%	48.89%	44.69%	43.01%	21.37%	43.13%	28.50%	**53.24%**
CH	**65.41%**	59.47%	44.09%	60.56%	31.24%	61.86%	**44.31%**	38.30%	21.88%	39.58%	8.57%	39.10%	**40.63%**	36.20%	17.59%	36.77%	14.11%	30.00%
Total	42.43%	40.95%	23.65%	38.85%	33.53%	**49.95%**	29.49%	27.19%	16.48%	25.11%	21.05%	**35.33%**	27.97%	28.40%	13.88%	26.26%	22.08%	**33.82%**

Our results show that across all code adequacy metrics, method-level prompting is substantially more effective than class-level prompting. Although both configurations are provided with the same source of information and the instruction to eventually target identical code, the difference in granularity was significant. We noticed that one contributing factor is the number of generated tests. Under class-level prompting, Plain-LLM generates 3,232 test cases, of which 50.53% pass successfully. In contrast, method-level prompting produces nearly four times as many tests, generating 12,158 test cases with a slightly higher passing rate of 53.40%.

Finding 3: Granularity plays a critical role in both the effectiveness and efficiency of LLM-based test generation. Method-level testing enables the model to explore methods more thoroughly by generating approximately four times as many tests, resulting in higher code coverage.

As expected, class-level testing requires significantly fewer LLM queries. Across the dataset, class-level testing generated a total of 1,510 tests, which is approximately 7.76 times fewer than the number generated at the method level.

VI. DISCUSSION

A. Generalization on multiple LLMs

To assess whether our findings generalize beyond the primary model under evaluation, we replicated our experiments using two additional large language models: DeepSeek V3 and LLaMA 3.3 (70B). These models were selected because they belong to different model families and represent state-of-the-art capabilities in large-scale code generation.

It is important to note, however, that the two models differ in their announced training data cutoff dates. LLaMA 3.3 reports a cutoff of December 2023, whereas DeepSeek V3 reports a substantially later cutoff of December 2024. As a result, DeepSeek V3 may have been exposed to the full set of projects used in our evaluation.

Table V summarizes the compilation and passing rates across the entire dataset for all evaluated models. Consistent with our earlier observations for GPT-4o-mini, method-level prompting yields substantially more generated tests than class-level prompting. For DeepSeek and LLaMA, we observe a modest improvement in passing rates; however, this comes at the cost of reduced compilation rates, indicating a trade-off between test correctness and syntactic validity in these models' outputs.

Table VI presents the effectiveness of each approach across all LLMs used. The trends observed previously are consistent across all models, supporting the generalizability of our findings. In particular, for DeepSeek V3, where the entire dataset may have been included in its training corpus, the Plain-LLM approach substantially outperforms all others. Although this cannot be confirmed definitively, the evidence suggests that, over time, the effectiveness of prior approaches may diminish.

Regarding efficiency, Figure 2 shows the number of requests of each approach to each LLM, and overall the trend looks similar. However, in the case of SymPromt, it clearly generated fewer tests with Llama 3.3 70B and DeepSeek-V3, which explains its degradation in terms of effectiveness.

We can also observe that the number of requests done by TestSpark, CoverUP, and the Plain-LLM (method and class granularity), remain consistent throughout the different LLMs. HITS, however, performed many more requests to the LLM when gpt-4o-mini is used, which explains why its performance is higher under this specific LLM.

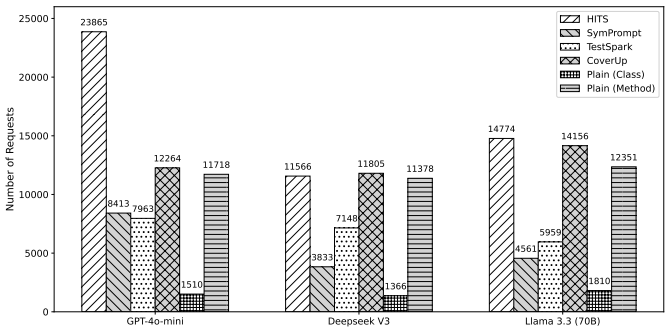

Fig. 2. Number of API requests between different approaches

Finding 4: Our findings generalize across multiple LLMs from different model families. Experiments with LLaMA 3.3 (70B) yield results consistent with those observed previously, and evaluations using DeepSeek V3 reveal an even larger performance gap between Plain-LLM and prior approaches.

B. Method and Class Granularity Complementarity

Although the generated class-level suites may look less effective than the method-level ones, we observed that these are indeed complementary. We merged the passing tests generated at the class-level and method-level from the previous experiment into a single test suite.

TABLE IV
COMPARISON OF CLASS-LEVEL AND METHOD-LEVEL TESTING

Project	Line Coverage			Branch Coverage			Mutation Score		
	Class Level	Method Level	Combined	Class Level	Method Level	Combined	Class Level	Method Level	Combined
BPG	36.47%	45.13%	**49.07%**	22.88%	32.60%	**36.05%**	29.33%	41.21%	**42.59%**
ER	35.58%	45.55%	**49.89%**	23.36%	32.10%	**35.71%**	23.03%	31.53%	**35.24%**
BC	15.21%	49.91%	**51.06%**	36.20%	49.77%	**53.39%**	7.85%	15.93%	**16.39%**
W	37.14%	36.96%	**44.72%**	24.90%	27.49%	**31.87%**	30.48%	32.81%	**38.88%**
CM	41.62%	65.39%	**69.08%**	25.56%	48.89%	**55.37%**	28.50%	53.24%	**57.90%**
CH	31.24%	61.86%	**63.67%**	8.57%	39.10%	**40.22%**	14.11%	30.00%	**31.71%**
Total	33.53%	49.95%	**53.67%**	21.05%	35.33%	**38.74%**	22.08%	33.82%	**36.55%**

As shown in Table IV, this combined suite consistently achieves higher coverage across all evaluated code coverage metrics. This observation indicates that the two approaches are complementary. While method-level prompting tends to produce more fine-grained and in-depth tests, class-level prompting enables the model to exercise code outside individual methods, thereby improving overall coverage.

Finding 5: Class-level and method-level workflows are complementary. Combining the benefits of the two, one can achieve higher coverage.

Nonetheless, combining class-level and method-level test suites requires a substantially higher cost in terms of generated tests and execution time, and increases the likelihood of duplicate tests. To address these limitations, we propose a *hybrid approach* that integrates the strengths of both workflows while improving efficiency. The *hybrid approach* initially applies class-level test generation and subsequently invokes method-level test generation only for methods that are not fully covered at the branch level. In other words, method-level testing is selectively applied to insufficiently covered methods.

Table VII compares the results of the combined and hybrid approaches showing that the observed differences are minimal. Statistical analysis using the Mann–Whitney U test [33] confirms that these differences are not statistically significant ($p = 0.92$ for branch coverage and $p = 0.75$ for line coverage). In other words, these results indicate that there is no difference in terms of effectiveness between combined and hybrid approach.

When it comes to efficiency though, our findings show that the hybrid approach is more optimized. In table VIII we observe that hybrid approach requires fewer queries to the LLM and used approximately half the number of tests compared to the combined approach. Furthermore, hybrid approach not only outperformed the combined test suite, but it was even more efficient than using only method-level testing.

TABLE V
COMPILATION AND PASSING RATES BETWEEN DIFFERENT LLMS

Name	Compilation Rate		Passing Rate	
	Plain (Class)	Plain (Method)	Plain (Class)	Plain (Method)
GPT 4o-mini	88.64% (2865/3232)	84.34% (10254/12158)	50.53% (1633/3232)	53.40% (6492/12158)
Deepseek V3	82.47% (3800/4608)	83.02% (16539/19922)	61.87% (2851/4608)	65.57% (13062/19922)
Llama 3.3 (70B)	65.56% (2640/4027)	79.14% (10477/13239)	55.55% (2237/4027)	57.30% (7586/13239)

Figure 3 depicts the number of queries issued by LLM-Plain to each respective model. Consistent with our previous observations, class-level prompting requires the fewest queries, whereas method-level and combined-level prompting necessitate substantially more. Notably, the Hybrid approach provides a favorable trade-off throughout the three LLMs, achieving improved coverage while reducing the total number of API requests.

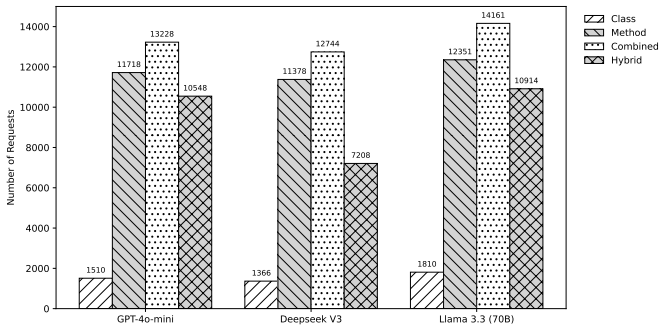

Fig. 3. Number of API requests for LLM-Plain across multiple LLMs

Finding 6: Hybrid approach achieves the same effectiveness as the combined class-level and method-level test suites, while exhibiting greater efficiency by requiring fewer LLM queries and generating fewer tests.

C. High number of non-compiling and failing test cases

Although all evaluated approaches aim to achieve full coverage, our results show that this goal remains largely out of reach. A central challenge across all approaches is the high number of non-compiling and failing test cases. As reported in Table V, a substantial portion of the generated tests either fail to compile or fail at runtime due to incorrect oracles.

In addition, based on our manual inspection, several failing cases cannot be attributed solely to incorrect oracles. In multiple instances, the failures stem from improperly mocked external dependencies. However, distinguishing between failures caused by incorrect assertions and those caused by an incorrect test prefix is difficult. To better understand these shortcomings, we conducted a detailed analysis of the generated test cases in order to identify common sources of failure.

TABLE VI
CODE COVERAGE BETWEEN DIFFERENT APPROACHES ACROSS ALL LLMS USED

		Line Coverage						Branch Coverage						Mutation Score					
Model	**Value**	**HITS**	**SYMP**	**TSP**	**CUP**	**Plain (C)**	**Plain (M)**	**HITS**	**SYMP**	**TSP**	**CUP**	**Plain (C)**	**Plain (M)**	**HITS**	**SYMP**	**TSP**	**CUP**	**Plain (C)**	**Plain (M)**
GPT-4o-mini	Total	42.43%	40.95%	23.65%	38.85%	33.53%	**49.95%**	29.49%	27.19%	16.48%	25.11%	21.05%	**35.35%**	27.97%	28.40%	13.88%	26.26%	22.07%	**33.81%**
Deepseek V3	Total	48.48%	33.74%	25.24%	42.10%	38.91%	**59.82%**	36.40%	22.09%	19.51%	32.98%	23.32%	**49.95%**	30.92%	16.05%	16.60%	28.17%	23.88%	**46.05%**
Llama 3.3 (70B)	Total	41.84%	28.56%	25.00%	36.61%	43.73%	**52.34%**	31.05%	16.38%	18.54%	25.43%	28.60%	**37.84%**	26.63%	13.37%	14.19%	20.51%	28.68%	**36.53%**

TABLE VII
COMPARISON OF COMBINED AND HYBRID RESULTS IN TERMS OF COVERAGE

Project	**Line Coverage**		**Branch Coverage**		**Mutation Score**	
	Combined	**Hybrid**	**Combined**	**Hybrid**	**Combined**	**Hybrid**
BPG	49.07%	49.47%	36.05%	38.56%	42.59%	44.27%
ER	49.89%	44.30%	35.71%	33.03%	35.24%	31.39%
BC	51.06%	47.86%	53.39%	54.30%	16.39%	17.10%
W	44.72%	43.98%	31.87%	31.87%	38.88%	37.17%
CM	69.08%	71.47%	55.37%	62.22%	57.90%	60.36%
CH	63.67%	63.98%	40.22%	40.14%	31.71%	35.70%
Total	53.67%	52.30%	38.74%	38.84%	36.55%	36.76%

TABLE VIII
COMPARISON OF COMBINED AND HYBRID IN TERMS OF # REQUESTS AND # TESTS (PASSING)

Project	# Requests		# Tests	
	Combined	**Hybrid**	**Combined**	**Hybrid**
BPG	2524	**2021**	2877	**1309**
ER	2525	**2310**	1149	**690**
BC	2749	**2682**	405	**336**
W	1703	**1309**	827	**464**
CM	2189	**1437**	980	**693**
CH	1538	**789**	1887	**834**
Total	13228	**10548**	8125	**4326**

1) Syntactically invalid test cases: In the implementation of ChatUniTest, which we use to execute the four state-of-the-art tools, non-compiling tests are filtered out by design if the execution is successful. For certain approaches, such as CoverUp, this behavior is unavoidable, as CoverUp requires a fully passing test suite to compute coverage for the class under test. In contrast, when running LLM-Plain, we are able to directly measure the number of non-compiling tests. This analysis reveals that a key limitation of several approaches lies in the generation of syntactically invalid test cases.

Table V reports the compilation and passing rates of LLM-Plain at both the class-level and method-level across all evaluated LLMs. On average, 78.89% of the generated tests successfully compile at the class level, compared to 82.17% at the method level. With respect to test execution, approximately half of the generated tests pass: on average, 55.98% at the class level and 59.42% at the method level. These results indicate that, despite LLMs' ability to generate test code, only about 40–50% of the generated tests are ultimately usable.

Finding 6: Plain-LLM demonstrates that one of the key reasons limiting the effectiveness of LLM-based test generation approaches is the frequent generation of syntactically and semantically invalid test cases. In our evaluation, only about half of the tests generated by LLM-Plain successfully passed.

Although not all corresponding studies explicitly report passing rates, the results that are available align with our findings. For instance, the passing rate reported for HITS closely matches the passing rate observed in our evaluation.

2) Code generation beyond unit tests: One of the earliest observations from our analysis was that a common cause of non-compiling or failing tests was the generation of additional classes or interfaces by the LLM. In several cases, instead of correcting the unit tests themselves, the LLM attempted to resolve compilation or runtime errors by introducing new classes or interfaces intended to override existing functionality. Hallucinations are a known prevalent issue on LLMs [34].

To better understand the prevalence of this behavior, we conducted a systematic analysis of the generated files and quantified the number of occurrences. Figure 4 illustrates both the number of instances in which the LLM generated additional classes or interfaces and the number of test files in which the LLM attempted to replace parts of the existing implementation. Such limitation shows that we need to find better test-repair mechanisms.

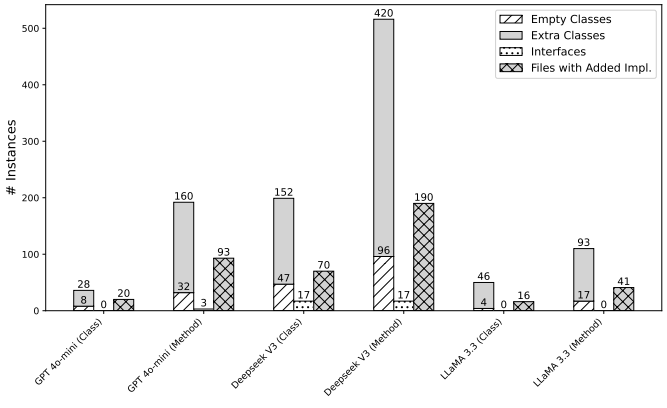

Fig. 4. Additional content generated by LLMs instead of fixing unit tests

Nonetheless, we observed another noteworthy behavior: the generation of *empty class*. Listing 1 presents an example of such an *empty class*. As shown, the generated class contains no functionality and appears to be introduced by the LLM merely as a placeholder.

```java
/** Helper classes for testing */
private static class TestClass {
    // Simple test class
}
```

Listing 1. Example of additional yet empty class generated by DeepSeek

```java
    // Test class for JSON deserialization
    static class User {
        [...]
        @Override
        public boolean equals(Object o) {
            if (this == o)
                return true;
            if (o == null || getClass() !=
                o.getClass())
                return false;
            User user = (User) o;
            return age == user.age && (name !=
                null ? name.equals(user.name) :
                user.name == null);
        }

        @Override
        public int hashCode() {
            int result = name != null ?
                name.hashCode() : 0;
            result = 31 * result + age;
            return result;
        }
    }
```

Listing 2. Example of additional generated by HITS

Figure 4 further quantifies this phenomenon, indicating that empty classes account for approximately 10–30% of all additionally generated classes.

Finding 7: LLMs tend to generate additional code beyond the intended unit tests. Across all evaluated models, we observed the generation of extra interfaces and auxiliary classes, as well as empty classes that appear to serve as placeholders.

Curious to see whether this phenomenon occurs in other approaches as well, we calculated all additional generated interfaces and classes in all approaches across the entire dataset. Our analysis confirms that all evaluated approaches produce extra content. Specifically, HITS, SymPrompt, CoverUp, TestSpark, and our implementation of LLM-Plain generated additional interfaces or classes in 6.01%, 5.14%, 3.89%, 6.24%, and 3.98% of all generated files, respectively. Listing 2 shows an additional class generated by HITS, which overrides existing content to assist on its generated tests. Figure 5 provides a detailed breakdown of the number of additional generated instances for each approach across the full dataset.

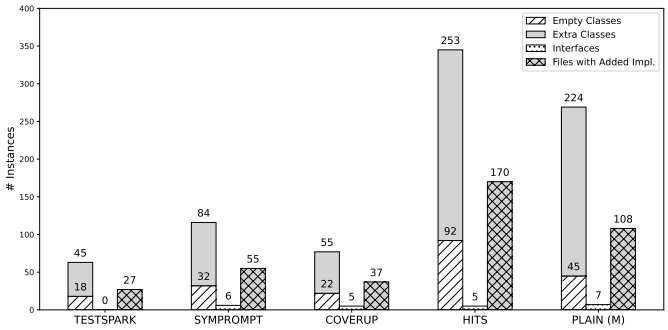

Fig. 5. Average additional content generated on whole dataset

Finding 8: All evaluated LLM-based approaches generated additional content beyond unit tests, which either overrides existing functionality, implements supplementary test classes, or serves as placeholders for developers to insert necessary code.

VII. Open Challenges

A. Fixing syntactically invalid tests

Our findings indicate that a substantial number of generated tests are discarded because they either fail to compile or do not pass, potentially due to incorrectly generated oracles. Averaging across all evaluated models, we found that at the class level, only 78.89% of tests successfully compiled, while on method level about 82,17%. In other words, an average of 854 tests per model were discarded at the class level, and 2,683 tests per model at the method level.

Although all approaches in our evaluation discard non-compiling tests, it is possible that an approach incorporating a test-repair mechanism could achieve improved results by correcting otherwise discarded tests.

Similarly, the prevalence of non-passing tests cannot be overlooked, as on average approximately half of the generated tests fail, thereby reducing the overall effectiveness of the generated test suite. Perhaps the primary reason for the failure of otherwise compiling tests appears to be incorrectly generated oracles. Using LLMs for test oracle generation is a research challenge in its own, with distinct advantages and limitations [35] [36]. Nonetheless, some approaches have already combined traditional test generation techniques with LLM-based oracle generation [37] [38]. A dedicated line of research that integrates an oracle generation mechanism into LLM-based test generation tools could substantially improve the quality and effectiveness of the generated tests.

B. Proper guidance: Avoid creativity and filling helpers

Based on our case study, we observed that a substantial number of classes could not be effectively tested using straightforward unit tests alone. Some classes required auxiliary helper classes to enable meaningful test execution. Although the LLMs occasionally attempted to generate such helper classes, these attempts were largely unsuccessful, leading us to exclude the generated helpers from the evaluation.

Moreover, beyond failing to produce functional helper classes, we found that approximately 10–30% of the generated classes were placeholders lacking concrete implementations and seemingly intended for manual completion by a developer. In other cases, the LLMs generated classes that overrode existing functionality rather than supporting it.

While overriding production functionality is generally undesirable, these observations suggest that automated test generation could benefit from the ability to synthesize accurate and well-scoped helper components. To the best of our knowledge, this aspect of LLM-based test generation has not yet been explored and represents a promising direction for future work.

C. Hybrid approach: Leveraging granularity

Our experiments show that granularity has a significant impact on effectiveness, as method-level prompting makes LLMs to generate more tests and cover a greater number of execution paths. At the same time, the two approaches (class-level and method-level) do not replace each other, but rather complement one another. However, while combining class-level and method-level approaches is effective, it is relatively cost-inefficient, requiring a high number of LLM queries and increasing the likelihood of generating duplicate tests.

In our case study, we observed that the Hybrid approach achieves effectiveness comparable to the combined approach while substantially reducing LLM query cost, despite our current implementation not being fully optimized. Moreover, whereas prior approaches in our evaluation primarily rely on method-level testing, our findings indicate that the Hybrid approach can be even more efficient than method-level testing alone. On average across all evaluated models, the Hybrid approach required 2,259 fewer LLM queries than method-level testing, which is the strategy employed by current approaches. These results highlight the need for further research on efficiently combining multiple levels of granularity to maximize effectiveness and efficiency.

VIII. Threads to Validity

External validity: A potential threat to the validity of this study concerns the tools used for comparison. To mitigate this risk, we relied on ChatUniTest, a widely adopted plugin that implements all the algorithms included in our evaluation. This tool has been employed in previous studies, including evaluations of LLM-based test generation approaches. In addition, we evaluated multiple LLMs from different model families to ensure that our findings are consistent and generalizable.

Internal validity: The performance of our implementation depends on the underlying LLMs. Accordingly, a potential threat to validity arises from variations in model capability as well as from the nondeterministic nature of LLM outputs. To mitigate these threats, we evaluated multiple models from different families, reducing the risk that our findings are specific to a single model. Additionally, to limit nondeterminism, we configured all models with a low temperature setting (0.1), aiming to make the generation process as deterministic as possible.

Construct validity: A potential threat to the validity of this study is data leakage, that is, whether the LLM has already been exposed to our evaluation set during training. To mitigate this threat, we selected repositories previously used in studies on LLM-based test generation [23] [5], as well as the GitBug-Java dataset [32]. Additionally, we manually inspected the creation date of each repository included in our experimental setup and labeled them accordingly to distinguish between seen and unseen data.

IX. Conclusion

In this study, we investigated the performance of existing LLM-based test generation approaches when evaluated on newer LLM versions than those used in their original assessments. We also implemented a simple *Plain-LLM* baseline that relies exclusively on the capabilities of the LLM, employing minimal output post-processing and no error handling or sophisticated engineering mechanisms.

Our findings indicate that simple prompting via Plain-LLM can outperform the four state-of-the-art tools included in our evaluation. Plain-LLM achieved higher total and average coverage than all prior approaches across most metrics, with the only exception being that HITS slightly outperformed Plain-LLM on average branch coverage. Regarding efficiency, Plain-LLM demonstrated comparable performance.

We further examined the impact of prompting granularity on LLM-based test generation. Our findings show that method-level prompting leads LLMs to generate a larger number of tests and to exercise methods more thoroughly than class-level prompting. To explore whether these two strategies can be combined, we evaluated their joint use and found them to be complementary. Building on this observation, we demonstrated that a hybrid approach that leverages both class-level and method-level workflows can improve the cost-effectiveness of test generation, outperforming method-level prompting alone as well as the naïve combination of their respective test suites.

Finally, we demonstrated that these results generalize across multiple LLMs of different families. In addition, we observed that LLM-based test generation techniques suffer from a large number of syntactical invalid tests, as under class level, only 78.89% of tests successfully compiled, while on method level about 82,17%.

X. Data availability

We provide a complete and executable implementation of LLM-Plain at the following link:

https://github.com/michaelkonstantinou/llm-plain

The repositories included in our dataset, as well as the state-of-the-art tools employed, are publicly available.

Acknowledgment

This work is supported by the Luxembourg National Research Fund (FNR), Grant number C22/IS/17426831/MeMoRIA; the FNR PEARL program Grant agreement 16544475; and the RDI Law project "Innovations for 21st Century Assessment Authoring," financed by the Luxembourg Ministry of the Economy.

REFERENCES

[1] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, Sep. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3695988

[2] C. Foster, A. Gulati, M. Harman, I. Harper, K. Mao, J. Ritchey, H. Robert, and S. Sengupta, "Mutation-guided llm-based test generation at meta," in *2025 ACM Conference on Foundations of Software Engineering (FSE 2025)*. ACM, 2025, also available as arXiv preprint arXiv:2501.12862.

[3] N. Alshahwan, J. Chheda, A. Finegenova, M. Harman, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using Large Language Models at Meta," in *ACM International Conference on the Foundations of Software Engineering (FSE 2024)*, July 2024.

[4] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660783

[5] Z. Wang, K. Liu, G. Li, and Z. Jin, "HITS: high-coverage llm-based unit test generation via method slicing," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, V. Filkov, B. Ray, and M. Zhou, Eds. ACM, 2024, pp. 1258–1268. [Online]. Available: https://doi.org/10.1145/3691620.3695501

[6] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Softw. Eng.*, vol. 50, no. 4, p. 911–936, Feb. 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3368208

[7] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. H. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *CoRR*, vol. abs/2201.11903, 2022. [Online]. Available: https://arxiv.org/abs/2201.11903

[8] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/forum?id=1PL1NIMMrw

[9] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, H. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, vol. 2, no. 1, 2023.

[10] J. A. Pizzorno and E. D. Berger, "Coverup: Coverage-guided llm-based test generation," *CoRR*, vol. abs/2403.16218, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.16218

[11] 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," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3643769

[12] 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.

[13] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, 1997. [Online]. Available: https://doi.org/10.1145/267580.267590

[14] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for java (demo)," in *ISSTA*. ACM, 2016, pp. 449–452.

[15] R. Pan, M. Kim, R. Krishna, R. Pavuluri, and S. Sinha, "Aster: Natural and multi-language unit test generation with llms," *arXiv preprint arXiv:2409.03093*, 2024.

[16] R. Liu, Z. Zhang, Y. Hu, Y. Lin, X. Gao, and H. Sun, "Llm-based unit test generation for dynamically-typed programs," *CoRR*, vol. abs/2503.14000, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2503.14000

[17] W. Wei, "Static analysis and llm for comprehensive java unit test generation," in *2025 8th International Conference on Advanced Electronic Materials, Computers and Software Engineering (AEMCSE)*, 2025, pp. 87–92.

[18] Q. Xu, G. Wang, L. C. Briand, and K. Liu, "Hallucination to consensus: Multi-agent llms for end-to-end test generation with accurate oracles," *CoRR*, vol. abs/2506.02943, 2025.

[19] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Trans. Software Eng.*, vol. 50, no. 1, pp. 85–105, 2024.

[20] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," in *SIGSOFT FSE Companion*. ACM, 2024, pp. 185–196.

[21] Z. Nan, Z. Guo, K. Liu, and X. Xia, "Test intention guided llm-based unit test generation," in *ICSE*. IEEE, 2025, pp. 1026–1038.

[22] C. Ni, X. Wang, L. Chen, D. Zhao, Z. Cai, S. Wang, and X. Yang, "Casmodatest: A cascaded and model-agnostic self-directed framework for unit test generation," *CoRR*, vol. abs/2406.15743, 2024.

[23] 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*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 572–576. [Online]. Available: https://doi.org/10.1145/3663529.3663801

[24] S. Gu, C. Fang, Q. Zhang, F. Tian, J. Zhou, and Z. Chen, "Testart: Improving llm-based unit test via co-evolution of automated generation and repair iteration," *CoRR*, vol. abs/2408.03095, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.03095

[25] S. Gu, N. Nashid, and A. Mesbah, "LLM test generation via iterative hybrid program analysis," *CoRR*, vol. abs/2503.13580, 2025.

[26] C. Yang, J. Chen, B. Lin, J. Zhou, and Z. Wang, "Enhancing llm-based test generation for hard-to-cover branches via program analysis," *CoRR*, vol. abs/2404.04966, 2024.

[27] P. Straubinger, M. Kreis, S. Lukasczyk, and G. Fraser, "Mutation testing via iterative large language model-driven scientific debugging," in *IEEE International Conference on Software Testing, Verification and Validation, ICST 2025 - Workshops, Naples, Italy, March 31 - April 4, 2025*. IEEE, 2025, pp. 358–367. [Online]. Available: https://doi.org/10.1109/ICSTW64639.2025.10962485

[28] G. Wang, Q. Xu, L. C. Briand, and K. Liu, "On mutation-guided unit test generation," *CoRR*, vol. abs/2506.02954, 2025.

[29] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," *Inf. Softw. Technol.*, vol. 171, p. 107468, 2024. [Online]. Available: https://doi.org/10.1016/j.infsof.2024.107468

[30] C. Foster, A. Gulati, M. Harman, I. Harper, K. Mao, J. Ritchey, H. Robert, and S. Sengupta, "Mutation-guided llm-based test generation at meta," *CoRR*, vol. abs/2501.12862, 2025.

[31] M. S. Bouafif, M. Hamdaqa, and E. Zulkoski, "PRIMG : Efficient llm-driven test generation using mutant prioritization," *CoRR*, vol. abs/2505.05584, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2505.05584

[32] A. Silva, N. Saavedra, and M. Monperrus, "Gitbug-java: A reproducible benchmark of recent java bugs," in *MSR*. ACM, 2024, pp. 118–122.

[33] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[34] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Trans. Inf. Syst.*, vol. 43, no. 2, Jan. 2025.

[35] F. Molina, A. Gorla, and M. d'Amorim, "Test oracle automation in the era of llms," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–24, 2025.

[36] M. Konstantinou, R. Degiovanni, and M. Papadakis, "Do llms generate test oracles that capture the actual or the expected program behaviour?" *arXiv preprint arXiv:2410.21136*, 2024.

[37] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2130–2141.

[38] S. B. Hossain and M. Dwyer, "Togll: Correct and strong test oracle generation with llms," *arXiv preprint arXiv:2405.03786*, 2024.