Improving LLM-Driven Test Generation by Learning from Mocking Information

Jamie Lee*, Flynn Teh*, Hengcheng Zhu†, Mengzhen Li‡, Mattia Fazzini‡, Valerio Terragni*

*_University of Auckland_, Auckland, New Zealand

†_The Hong Kong University of Science and Technology_, Hong Kong SAR

‡_University of Minnesota_, Minneapolis, USA

Emails: *{eejl773, fteh492, vter674}@aucklanduni.ac.nz, †hzhuaq@connect.ust.hk, ‡{li001618, mfazzini}@umn.edu

***Abstract*—Large Language Models (LLMs) have recently shown strong potential for automated unit test generation. This has motivated us to investigate whether developer-defined test doubles (commonly referred to as mocks) available in existing test suites can be leveraged to improve LLM-driven test generation. To this end, we propose MOCKMILL, an LLM-based technique and tool that generates test cases by exploiting mocking information automatically extracted from developer-written tests. MOCKMILL targets components that are replaced by test doubles in existing tests and uses the encoded stubbings and interaction expectations to guide test generation, combined with an iterative generation-and-repair process to ensure executable tests. We evaluated MOCKMILL on 10 open-source classes from six JAVA projects using four LLMs, and compared the generated tests with existing project tests and tests produced by baseline approaches. The results show that MOCKMILL's tests cover lines of code and kill mutants that existing tests and baseline-generated tests miss. Overall, our findings provide preliminary evidence that leveraging mocking information is a complementary and effective way to enhance LLM-based test generation.**

***Index Terms*—Test Generation, LLMs, Test Doubles, Mocking.**

I. INTRODUCTION

Software underpins modern life, making rigorous testing essential to ensure correctness [1]. Yet writing high-quality tests is labor-intensive [2], motivating decades of automated test generation research [2], [3]. Long-standing approaches range from random testing to search-based methods [4]. Recently, the emergence of Large Language Models (LLMs) has opened a new frontier for automated test generation [5], [6], [7]. LLMs, trained on vast code corpora, can produce useful tests, often with minimal human guidance [8], [9]. This capability has sparked interest in LLM-based testing [9]. However, while the results of LLM-driven test generation are highly promising, many aspects of LLM-based test generation remain underexplored, particularly the impact of providing specific auxiliary artifacts beyond the class under test [9], [8].

In this paper, we explore how to improve LLM-driven test generation by leveraging *test doubles* [10] (or more informally also referred to as *mocks*). Test doubles are test artifacts commonly used to manage the complexity of interacting components during testing. They are lightweight stand-in objects that mimic real components, allowing developers to isolate the software under test from its dependencies [10], also referred to as *mocked components*. Test doubles can be configured to return predefined responses for specific inputs (*stubbings*) or to check expected interactions (*verify operations*). Through them, developers implicitly specify how dependent components should behave. Test doubles are central to modern software testing [10], and studies across programming languages show that stubbing and verify operations are widely used in both industry and open-source software [11], [12], [13], [14]. Tests generated from test double information can help verify dependent components when they are part of the software system.

Existing LLM-based approaches typically generate tests using the code under test and its associated documentation or comments [8], [9], making the rich usage patterns and expected behaviors encoded in test doubles an untapped resource for improving automated test generation. To bridge this gap, we introduce **MOCKMILL**, an *LLM-based approach that leverages the mocking information from an existing test suite to generate new tests*. Given the software under test and its test suite as inputs, MOCKMILL generates tests for those components that are replaced by test doubles in existing tests. The approach starts by automatically extracting mocking information from the test suite via static analysis. MOCKMILL then employs an LLM-guided approach to generate tests for targeted components. The approach integrates a generation-and-repair loop: the extracted mock information is incorporated into the LLM input to generate candidate tests, and any compilation or runtime issues in the tests are iteratively fixed via LLM-based corrections. By guiding an LLM with this existing, developer-defined information, MOCKMILL produces tests that align with the behavioral expectations already encoded in the test suite.

We implemented MOCKMILL and evaluated it on 10 classes from six open-source JAVA projects using four LLMs (GPT–4O MINI, GPT–5 MINI, GPT–5, and CLAUDE SONNET 4.5). We compare the tests generated by MOCKMILL with existing project tests and with two baselines: an LLM-based approach that does not utilize mocking information and random test generation (RANDOOP) [15]. We measured line and mutation coverage to assess effectiveness. The results suggest that MOCKMILL covers code and kills mutants missed by existing and baseline tests, providing preliminary evidence that it complements existing or automatically generated tests. We released MOCKMILL's code and experimental data [16].

This is the authors' version of the paper published in IEEE International Conference on Software Testing and Verification Workshops (ICSTW) - International Workshop on Artificial Intelligence in Software Testing (AIST).

II. INTUITION AND RUNNING EXAMPLE

Our key **intuition** is that *mocking information encodes meaningful knowledge about how software components should be used and this can be leveraged to create useful tests*. This work specifically leverages *stubbings* and *verify operations* [17] as *mocking information* (or *mocking data*), which often specify: (i) which methods of dependent components are invoked, (ii) the arguments passed to the methods, and (iii) the return values or exceptions resulting from those invocations. Stubbings are the natural way to encode (i), (ii), and (iii). Furthermore, verify operations can also specify (i) and (ii) by checking that certain methods have been called with certain arguments during test execution [17]. Such structured behavioral information serves as *implicit usage documentation of dependent components*. We use *dependent components*, *replaced components*, *mocked components* interchangeably to refer to those components that are replaced by test doubles and on which stubbings and verify operations are defined. When provided to an LLM during test generation, the information can guide the model toward realistic and meaningful tests that improve the coverage of the test suite. MOCKMILL exploits this insight by extracting mocking information from existing tests and using the information to generate tests for the mocked components. These components are the *target components* in the test generation process and are the dependent components in existing tests.

Figure 1 shows a **running example**, illustrating how MOCKMILL leverages mocking information and the advantage of incorporating it. The figure includes three tests: the original developer-written test `pageQueryAopLogsTest` (from which MOCKMILL extracts the mocking data), the test generated by an LLM-driven baseline without providing mocking information (`baseline_test`), and the test generated by MOCKMILL using the extracted mocking data (`mockmill_test`). `pageQueryAopLogsTest` contains mocking data for the `AopLogRepository` class (i.e., the dependent component). This component is the target component (i.e., target class) for `baseline_test` and `mockmill_test`. `AopLogRepository` is also a subject of our evaluation (Section IV).

```
1 // Developer-written test with mocking info
2 @Test
3 void pageQueryAopLogsTest() {
4   ... aopLogRepository = mock(AopLogRepository.class);
5   PageRequestDto pageDto = PageRequestDto.of(1, 10);
6   AopLogQueryDto queryDto = new AopLogQueryDto();
7   when(aopLogRepository.pageFetchBy(pageDto, queryDto)).
        thenReturn(mockResult);
8   // ...}
9
10 // Baseline-generated test without mocking info (line 7)
11 @Test
12 void baseline_test() {
13   AopLogRepository repository = new AopLogRepository();
14   AopLog aopLog = new AopLog();
15   aopLog.setId(1L);
16   repository.insert(aopLog);
17   AopLogQueryDto queryDto = mock(AopLogQueryDto.class);
18   when(queryDto.getId()).thenReturn(2L);
19   List<AopLog> result = repository.fetchBy(queryDto);
20   assertTrue(result.isEmpty());}
21
22 // MockMill-generated test giving mocking info (line 7)
23 @Test
24 void mockmill_test() {
25   AopLogRepository aopLogRepository = new AopLogRepository
        ();
26   PageRequestDto pageRequestDto = PageRequestDto.of(0,10);
27   AopLogQueryDto queryDto = new AopLogQueryDto();
28   Result<Record> mockResult = mock(Result.class);
29   Result<Record> result = aopLogRepository.pageFetchBy(
        pageRequestDto, queryDto);
30   assertEquals(mockResult, result);}
```

Fig. 1. Running example based on the `AopLogRepository` class.

The baseline approach generated a test that exercises only the `fetchBy` method, representing the most common and straightforward usage of the `repository` object. The existing developer test `pageQueryAopLogsTest` mocks `AopLogRepository` and stubs its `pageFetchBy` method, which reflects a more advanced usage scenario involving paging. Providing the mocking information in `pageQueryAopLogsTest` enables MOCKMILL to guide an LLM to generate a test that exercises the paging mechanism of `AopLogRepository`, thereby covering more lines of code and killing mutants injected in that method.

III. MOCKMILL

MOCKMILL is an LLM-based automated test generation approach that leverages mocking information to generate test cases. Its methodology comprises four phases: (1) project analysis, (2) **mock extraction** (introduced in this work), (3) test generation, and (4) post-generation repair. Phases (1), (3), and (4) follow common patterns in prior LLM-assisted test generation approaches [8], [9], while phase (2) is novel and extracts realistic usage information about mocked components that prior approaches do not explicitly leverage.

Figure 2 provides a high-level overview of MOCKMILL. MOCKMILL takes as input a software project configured with a standard build automation tool (e.g., MAVEN or GRADLE for JAVA). It also accepts a configuration file to filter target components (e.g., classes in JAVA), specify LLM settings, define the number of repair attempts, and set token limits. MOCKMILL begins with the Project Analysis phase, which finds existing tests using stubbings or verify operations and identifies associated components, which are the target components for test generation. The Mock Extraction phase then gathers structural data (i.e., the specific content of stubbings and verify operations) about the dependent component. This data is passed to the Test Generation phase, where the LLM produces candidate test cases based on extracted mock data. Finally, the Post-Generation Repair phase compiles, executes, and iteratively repairs the generated tests until they run successfully. The output of MOCKMILL consists of automatically generated tests for components that are substituted by test doubles in other tests. The output also includes logs about compilation and repair attempts, along with code and mutation coverage reports that help assess the quality of the generated tests. The rest of this section describes MOCKMILL's phases in detail. The LLM prompts used by the approach are omitted due to space limitations but are available in the replication package [16].

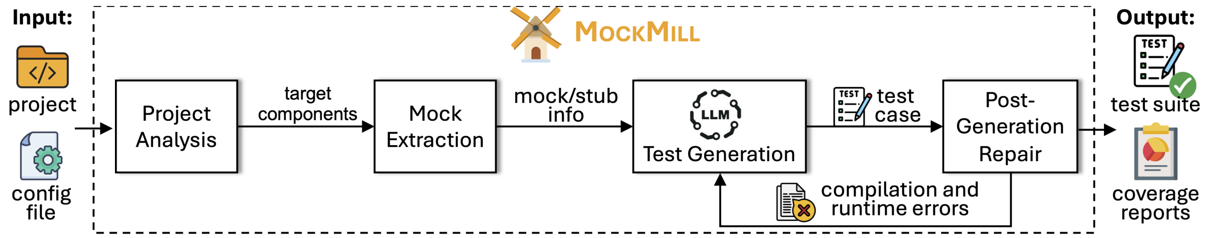

Fig. 2. High-level overview of MOCKMILL's workflow.

A. Project Analysis

The Project Analysis phase identifies the dependent components that can be targets for test generation by analyzing the test code in the project under analysis. Specifically, this phase parses the abstract syntax tree (AST) of the test files in the project to identify the components that have been replaced by test doubles and that use stubbings or verify operations. In our implementation, we identified the test cases written using JUNIT 5 and the test doubles created with MOCKITO, since they are the most popular testing and mocking frameworks in the Java ecosystem [14]. Once MOCKMILL identifies this information, this phase filters out components that are not part of the project (i.e., components from third-party libraries) by keeping only the classes that are defined in the source code of the project (and not in the libraries used by the project). The approach performs this step as generated tests should focus on code that belongs to the project. MOCKMILL encodes the relevant targets in a structured representation (i.e., a `JSON` file) that is then passed to the subsequent phase of the approach.

Running Example: For the project in Figure 1 (lines 2–9), this phase identifies that the test `pageQueryAopLogsTest` mocks `AopLogRepository` and defines a stubbing on it, selecting the class as a candidate for mock-informed test generation.

B. Mock Extraction

The Mock Extraction phase focuses on collecting the mocking information only relevant to each target component and structuring it into an intermediate JSON representation (which will be provided to the LLM for test generation). This phase identifies calls to mocking APIs of the supported mocking frameworks (e.g., `when(...).then(...)` or `verify()` in MOCKITO), along with argument values associated with the calls to the API methods by parsing the AST of the test files. The analysis traverses both test and setup methods to record field assignments, verification calls, and stubbing logic associated with the selected class. This focused extraction ensures that the Test Generation phase receives precise input data without including the entire source files of the test, which could otherwise overload the LLM context window and add noise, reducing the focus and overall quality of generated tests.

Running Example: In Figure 1, the Mock Extraction phase analyzes `pageQueryAopLogsTest` and records that the method `pageFetchBy(pageDto, queryDto)` of `AopLogRepository` is stubbed. This extracted detail becomes part of the structured input provided to the generator, ensuring that the LLM is aware of the paging behavior exercised in the developer-written test.

C. Test Generation

The Test Generation phase constructs LLM prompts for automated test creation based on the target component and the extracted mock information. In this phase, a structured prompt is assembled that provides the LLM with (i) explicit test generation instructions, (ii) the complete source code of the target component (e.g., the source code of the target class), and (iii) the data associated with the relevant stubbings and verify operations. Including the full class source code enables the LLM to reason about the implementation context, dependencies, and behaviors, thereby reducing the likelihood of incorrect assumptions or hallucinations [5], [6].

Prompt design: The test generation prompt targets mutation-based adequacy [18], as mutation coverage is widely recognized as one of the most comprehensive test adequacy criteria [18]. The prompt instructs the LLM to (i) use exact values from stubbings and verify operations; (ii) create precise assertions that fail under realistic mutations (bugs); (iii) test both true and false execution paths and boundary conditions derived from mocking information; and (iv) generate tests that instantiate and exercise real objects of the CUT rather than substituting them with test doubles.

Baselines. The approach also uses a few-shot prompting strategy [19] and provides two examples for the requested task. The examples are based on simple, self-contained JAVA classes along with corresponding structured mock and stub data extracted from representative tests. This data captures real interaction patterns that reflect how the CUT is used. The examples then include executable unit tests that instantiate the CUT, exercise its behavior, and assert on expected outputs. We intentionally designed the two few-shot examples to be domain-agnostic (and are not based on any of the projects considered in the evaluation). The prompt also does not target or identify specific mutants to kill [20]. It encourages the LLM to produce tests that satisfy mutation-based adequacy criteria in a general sense. Finally, the prompt specifies the required output format, directing the LLM to produce a compilable test file that includes all necessary package declarations and import statements.

TABLE I
EVALUATION DATASET OF OUR EXPERIMENTS. THE TABLE REPORTS THE DETAILS ON THE TARGET COMPONENTS (CUTS) AND THEIR PROJECTS.

CUT ID	Name of CUT	Project Name	Project LOC	CUT LOC	Methods	Max CC	Tests w/ TDs		Ss	VOs	Dev Tests?
							w/ Ss	w/ VOs			
CAS	`CustomerApplicationService`	E-Commerce Platform	45.9K	209	13	9	5	5	8	9	✗
RRA	`ResourceRightSizingAnalyzer`	E-Commerce Platform	45.9K	336	15	13	4	0	6	0	✓
MAS	`McpAsyncServer`	MCP Java SDK	27.4K	100	8	13	20	11	23	12	✗
MUT	`McpUriTemplateValidator`	MCP Java SDK	27.4K	271	16	10	8	5	16	6	✗
MQS	`MaestroQueueSystem`	Maestro	85.7K	125	6	8	1	15	1	16	✓
MPC	`ModulePermissionConverter`	Miaocha	37.0K	205	8	12	2	2	4	4	✗
QPC	`QueryPermissionChecker`	Miaocha	37.0K	150	6	6	5	5	5	5	✓
SMS	`SemanticSchema`	SuperSonic	61.5K	130	19	7	1	0	2	0	✓
ALR	`AopLogRepository`	ZhiLu AI Management	7.5K	92	7	22	5	5	5	5	✓
URE	`UserRepository`	ZhiLu AI Management	7.5K	97	7	10	3	3	3	3	✗

D. Post-Generation Repair

After generation, MOCKMILL iteratively compiles and repairs tests, following prior LLM-driven test generation [21], [8]. If compilation fails, the resulting error messages are passed to the LLM with targeted repair instructions to produce a corrected test file. This cycle repeats until the tests compile successfully or the retry limit is reached. The same iterative process applies to runtime failures. Once compilation succeeds, MOCKMILL executes the tests and, upon failure, provides the error logs to the LLM for correction. This loop continues until all tests pass or the maximum retry limit is reached.

This design assumes a regression testing scenario [22], [23], where failing generated tests are attributed to issues in the tests themselves rather than bugs in the target component. By default, MOCKMILL treats failures as test issues (e.g., a component not properly set up) and attempts to fix them. Users can also manually inspect failing tests if they suspect the failure originates from bugs in the target component. The final test suite is executed for coverage and mutation analysis to evaluate adequacy and fault-detection capability.

Prompt design: The repair prompt includes the error message along with the source code of the target component and the generated tests. It explicitly forbids explanations or comments and requires the LLM to use a significant change in approach on repeated attempts. These constraints reduce verbosity and focus the LLM on producing concise and executable tests.

IV. EVALUATION

This section presents the evaluation of MOCKMILL, which is based on the following research questions (RQs):

RQ1: Effectiveness – *To what extent can* MOCKMILL *generate effective tests?*

RQ2: Complementarity – *To what extent do tests generated by* MOCKMILL *complement those generated by baseline approaches and existing tests?*

RQ3: Cost – *What is the cost of generating tests using* MOCKMILL*?*

To evaluate MOCKMILL, we implemented it in a prototype tool for JAVA projects that use JUNIT [24] and MOCKITO [25]. It supports both MAVEN and GRADLE-based projects for automated compilation and test execution. We implemented the static analysis and mock extraction of MOCKMILL using the JAVAPARSER (https://javaparser.org/) library, which enables AST-based analyses of source and test files. All model calls and repair iterations are logged to facilitate reproducibility and further analysis.

A. Dataset

We evaluated MOCKMILL on open-source repositories selected from GitHub based on the following criteria. The repository uses at least JAVA 8, JUNIT 5, and MOCKITO 4 to ensure compatibility with modern testing practices. To mitigate data leakage, we included repositories updated after the latest knowledge cutoff date of the model considered (January 2025). The repository has at least 20 GitHub stars, indicating community interest and maintenance. The repository uses MAVEN or GRADLE to facilitate automated building, dependency resolution, and test execution. The repository contains at least one mocked class.

Mining GitHub using the above criteria yielded 24 candidate repositories. We manually examined each repository to identify target components (which we also refer to as components under test or CUTs) based on the following criteria. The CUT has at least 50 lines of code, ensuring sufficient implementation complexity. The CUT implements at least five methods, with at least one having cyclomatic complexity ≥ 5, as done in related work [4]. The CUT is replaced by a test double with stubbings or verify operations in other tests, allowing MOCKMILL to extract and reuse its defined mocking behavior.

This selection methodology yielded ten CUTs from six unique repositories. Table I provides details on the projects and the CUTs. Column "Max CC" denotes the highest cyclomatic complexity of the methods in the CUT. Columns under "Tests w/ TDs" report the number of tests that define a stubbing ("w/ S") or a verify operation ("w/ VO) on the test doubles replacing the CUTs. Column "Dev Tests" indicates whether the repository contains developer-written tests that directly test (i.e., exercise) the CUTs.

TABLE II
LLMS USED IN THE EVALUATION AND TOKEN COSTS.

Provider	Model	Input Cost	Output Cost
OPENAI	GPT–4O MINI	$0.15 / 1M tok	$0.60 / 1M tok
OPENAI	GPT–5 MINI	$0.25 / 1M tok	$2.00 / 1M tok
OPENAI	GPT–5	$1.25 / 1M tok	$10.00 / 1M tok
ANTHROPIC	CLAUDE SONNET 4.5	$3.00 / 1M tok	$15.00 / 1M tok

B. Methodology

Baselines. To evaluate the complementarity of MOCKMILL, we considered three baselines: an LLM-based test generation approach (which we refer to as LLM), RANDOOP [15] (i.e., an approach based on random testing), and the developer-written tests. LLM uses the same prompt and few-shot examples as MOCKMILL but omits mock and stub information. This configuration allows for a controlled comparison that isolates the relative contributions of mock information in the LLM-driven test generation process. Although EVOSUITE is generally more effective than RANDOOP [4], we used RANDOOP because it provided better support for the recent JAVA versions associated with the projects considered. Across projects, developer-written tests were found for only five CUTs.

Models. In RQ1, we evaluated MOCKMILL in combination with four LLMs. Table II lists the models we used and the token costs used at the time we ran the experiments (September 2025). At that time, GPT–4O MINI was recognized as a lightweight, cost-efficient model, GPT–5 MINI and GPT–5 represented newer reasoning-capable variants, and CLAUDE SONNET 4.5 provided a cross-provider comparison. All models were used with default parameters. In RQ2, we used GPT–5 MINI, as it was a top-performing model in RQ1 and it was cheaper than the other top-performing model CLAUDE SONNET 4.5. Selecting this model allowed us to focus the comparison on the relative benefits of MOCKMILL while controlling experimental costs and maintaining consistency across runs.

Metrics. We considered four classes of metrics: *generation*, *compilation*, *execution*, and *quality*. In terms of *generation*, we report the average number of *TeSt generaTed per CUTs* (TST). *Compilation* metrics assess whether MOCKMILL produces tests that can be built automatically. These metrics capture feasibility boundaries: percentage of tests that *Compiles on the First Try* (CFT) demonstrates immediate usability, while percentage of tests that *Compiles EVentually* (CEV) reflects the effectiveness of MOCKMILL's repair loop. *Execution* metrics measure how many executable tests MOCKMILL generates. *TestS Passed* (TSP) reports the percentage of tests that pass. *Quality* metrics assess how effectively the generated tests detect faults and exercise code, since quantity and executability alone do not ensure usefulness. We report the minimum (MIN), median (MED), maximum (MAX), and standard deviation (STDEV) of mutation score (fault detection) and line coverage (structural coverage) [26]. We use PIT [27] (with all mutations enabled) and JACOCO [28] to measure mutation score and line coverage, respectively. For fault detection, we also report *Unique Mutations Killed*, which isolates the exclusive contributions of a given technique, revealing complementarity that aggregate percentages can hide. For structural coverage, *Unique Lines Covered* highlights lines reached only by a technique. Together, these metrics provide a nuanced picture, as it is possible to identify when approaches add non-overlapping value relative to others.

Setup. We executed MOCKMILL and LLM ten times to account for randomness in LLM outputs. We repeated the same procedure for RANDOOP.

C. Results

1) ***RQ1: Effectiveness****: – To what extent can* MOCKMILL *generate effective tests?* Table III reports the results associated with RQ1. The average number of generated tests (TST) varies markedly across models. While GPT–4O MINI, GPT–5 MINI, and GPT–5 generate between 8.5 and 13.2 tests on average, CLAUDE SONNET 4.5 produces a much larger number of tests (45.7 on average). However, this larger test volume does not translate into clearly better quality than other models in terms of median mutation score and line coverage.

MOCKMILL is generally successful at producing compilable tests across all considered models. Although first-try compilation (CFT) varies across models, eventual compilation (CEV) is consistently high, ranging from 92% for GPT–4O MINI to 100% for the remainder. This result indicates that the generation-and-repair loop of MOCKMILL is effective at overcoming a substantial portion of the issues that arise during initial test generation.

The execution results further support the effectiveness of MOCKMILL. In terms of pass rate, all models except GPT–4O MINI achieve near-perfect results, with TSP values between 98.6% and 99.7%. Taken together, these results indicate that MOCKMILL is able to generate not only compilable but also executable tests with high reliability, especially when paired with GPT–5 MINI, GPT–5, or CLAUDE SONNET 4.5.

The mutation score results show that MOCKMILL can generate tests with strong fault-detection capability. All models reach high maximum mutation scores, ranging from 85% for GPT–4O MINI to 100% for GPT–5, GPT–5 MINI, and CLAUDE SONNET 4.5. The median mutation score provides a more robust picture of typical performance across runs. Here, GPT–5 MINI and CLAUDE SONNET 4.5 obtain the strongest medians, with 84% and 89%, respectively, while GPT–5 reaches 62% and GPT–4O MINI 43%. At the same time, the standard deviation values are relatively large for all models, indicating variability across runs and CUTs. Still, the high medians and maxima for GPT–5 MINI, GPT–5, and CLAUDE SONNET 4.5 show that MOCKMILL can often generate tests that kill a substantial fraction of mutants. Among these models, GPT–5 MINI stands out because it combines a high median mutation score with much stronger compilation and execution behavior than CLAUDE SONNET 4.5 (and at much lower cost, see RQ3).

The line coverage results further support the effectiveness of MOCKMILL. Median line coverage is high for the three

TABLE III
EVALUATION RESULTS OBTAINED BY RUNNING MOCKMILL ON THE CUTS OF THE DATASET (RQ1).

Model	Generation	Compilation		Execution	Mutation Score (%)				Line Coverage (%)			
	TST	CFT (%)	CEV (%)	TSP (%)	MIN	MED	MAX	STDEV	MIN	MED	MAX	STDEV
GPT–4O MINI	8.5	55	92	81.6	0	43	85	27.0	0	58	93	33.4
GPT–5	13.2	43	100	98.6	16	62	100	26.7	79	91	100	6.4
GPT–5 MINI	11.4	88	100	99.7	3	84	100	25.6	27	93	100	11.4
CLAUDE SONNET 4.5	45.7	44	100	99.7	5	89	100	33.1	39	94	100	12.7

strongest models: 91% for GPT–5, 93% for GPT–5 MINI, and 94% for CLAUDE SONNET 4.5. Moreover, all three models reach 100% maximum line coverage, while GPT–4O MINI reaches a maximum of 93%. GPT–4O MINI shows substantially weaker typical performance, with a median line coverage of 58%, whereas the other three models consistently exercise a large portion of the target code. Although CLAUDE SONNET 4.5 attains the highest median line coverage, its advantage over GPT–5 MINI is marginal (94% vs. 93%). Thus, considering line coverage together with compilation and execution reliability, GPT–5 MINI emerges as the most practical and effective choice.

Overall, the results provide evidence that MOCKMILL is effective at generating useful tests. Across the stronger models, MOCKMILL almost always produces executable tests after repair and often achieves high line coverage and mutation scores, indicating that the generated tests are not merely runnable but also meaningful from a testing perspective. Among the evaluated models, GPT–5 MINI offers the best overall trade-off. It achieves the highest first-try compilation rate, perfect eventual compilation, near-perfect pass rate, and strong median mutation score and line coverage. At the same time, it is considerably smaller and cheaper than GPT–5 and CLAUDE SONNET 4.5, making it the most effective and practical model for MOCKMILL based on our evaluation.

2) ***RQ2: Complementarity*** *– To what extent do tests generated by* MOCKMILL *complement those generated by baseline approaches and existing tests?:* Across CUTs, MOCKMILL achieves higher unique mutation kill rates than both LLM and RANDOOP only (shown in Table IV). CAS and RRA are prime examples of this, showing a MOCKMILL detection rate of 24.56% and 25.64%, compared to LLM (0-12.82%). MOCKMILL achieves a strictly higher unique mutation killed rate than LLM for 40% of CUTs and RANDOOP for 50% of CUTs. RANDOOP rarely kills any unique mutations, with killing 2.56% only on RRA. For MAS, MUT, QPC, and SMS, the table shows 0% across all approaches, indicating that MOCKMILL performs similarly across all cases. This suggests that mock information generally helps identify unique mutation cases missed by other tools. The low RANDOOP performance indicates that a simple traditional test generation approach might struggle to detect nuanced faults without the contextual and semantic reasoning available to LLMs. Unique line coverage is generally low across all CUTs, with only minor contributions from MOCKMILL (e.g., 2.61% for RRA, 1.56% for MQS, and 1.79% for URE) and one case for RANDOOP (2.61% for RRA). Mock information yields a small but measurable advantage, exposing a few additional lines not reached by other techniques. However, its overall impact on coverage is less pronounced than on mutant detection.

Developer tests contribute the fewest unique mutations (2.56% for RRA, none elsewhere), indicating that LLM-generated tests can reveal faults missed by human developers. Mock information enhances this advantage by improving fault detection diversity. For line coverage, differences are minor. MOCKMILL-generated tests again show the highest exclusive coverage (e.g., 2.61% for RRA, 1.56% for MQS), while no-mock and developer tests contribute little additional coverage. Although the differences are minor, there is evidence that mock information can help LLMs reach certain lines missed by LLM, RANDOOP, and developer tests.

3) ***RQ3: Cost*** *– What is the cost of generating tests using* MOCKMILL*?:* Table V presents the average test generation cost (USD) and input/output token usage for MOCKMILL and LLM across LLMs. Costs were derived from the input and output tokens consumed in each run and the corresponding model-specific API pricing.

At the prompting level, providing mock information (MOCKMILL vs. LLM) slightly increases input tokens and thus cost by increasing the prompt length. These effects typically add around 5-15% of cost within the same model, and start to be noticeable with models where token rates are higher.

At the model level, cost increases predictability with model size and capabilities. CLAUDE SONNET 4.5 has the highest cost, followed by GPT–5 (˜$0.08–$0.11), whereas the mini variants are quite efficient (typically under $0.02), with GPT–5 MINI being the most cost-effective model in terms of cost and quality of generated tests.

V. DISCUSSION

Interpretation of Results. The results indicate that MOCKMILL can guide LLM models toward useful tests. By leveraging mock information, MOCKMILL provides contextual cues that help uncover behaviors and faults missed by a vanilla LLM, RANDOOP, or developer-written tests. This is supported by the moderate number of unique mutations killed and unique lines of code covered by MOCKMILL. Because each experiment was repeated ten times, we are confident that the observed benefits are consistent and not due to random variation in LLM outputs. Mock-informed test generation consistently killed mutations that were otherwise undetected. This suggests that stubbings

TABLE IV
BASELINE COMPARISON WITH LLM, RANDOOP AND DEVELOPER-WRITTEN TESTS BY CUTS (RQ2)

CUT ID	Unique Mutations Killed (%)				Unique Lines Covered (%)			
	MOCKMILL	LLM	RANDOOP	Dev Tests	MOCKMILL	LLM	RANDOOP	Dev Tests
CAS	24.56	1.75	0.00	–	0.00	1.18	0.00	–
RRA	25.64	12.82	2.56	2.56	2.61	0.65	2.61	2.61
MAS	0.00	0.00	0.00	–	0.00	0.00	0.00	–
MUT	0.00	0.00	0.00	–	0.00	0.00	0.00	–
MQS	10.53	5.26	0.00	0.00	1.56	0.00	0.00	0.00
MPC	1.23	0.00	0.00	–	0.83	0.00	0.00	–
QPC	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ALR	4.26	4.26	0.00	0.00	0.00	0.00	0.00	0.00
URE	0.00	0.00	0.00	–	1.79	0.00	0.00	–

TABLE V
AVERAGE TEST GENERATION COST AND TOKEN USAGE (RQ3)

Model	MOCKMILL			LLM		
	Cost USD	Input Tokens	Output Tokens	Cost USD	Input Tokens	Output Tokens
GPT–4O MINI	$0.0076	36,761	3,506	$0.0062	26,805	3,632
GPT–5 MINI	$0.0176	16,477	6,723	$0.0171	13,352	6,852
GPT–5	$0.1028	15,228	8,377	$0.0786	9,043	6,733
CLAUDE SONNET 4.5	$0.5132	78,713	18,471	$0.4358	58,918	17,272

and verify operations activate distinct reasoning pathways within models and help them focus on different faults.

Complementary. We applied a *Kruskal–Wallis test* to assess whether mutation scores and line coverage differ significantly across techniques. All H values (0.19–1.94) and p-values ($\geq$ 0.584) indicate no statistically significant differences between MOCKMILL and the BASELINE. As such, MOCKMILL complements existing approaches. It can enhance existing test suites by generating additional tests that detect more faults and expand behavioral coverage. Indeed, the contextual cues derived from stubbing and verify operations that guide test generation may reduce test diversity, since the model tends to follow the provided interaction patterns. Future LLM-driven test generation could consider combining prompts both with and without mock information when available.

Cost and Practical Implications. The cost analysis shows that MOCKMILL incurs only a modest cost overhead (about 5–15% higher than LLM generation without mock information) while killing unique bugs and covering additional code. This trade-off is favorable for practical adoption, as MOCKMILL can deliver measurable testing benefits at an affordable additional computational cost.

A. Threats to Validity

Generalizability. We evaluated our approach on ten classes from six projects. Future work is needed to assess whether the approach generalizes across other projects, languages, and mocking frameworks. Although the current prototype and experiments focus on MOCKITO and JAVA-based projects, the approach can be extended to other languages and frameworks. For example, to PYTHON projects using UNITTEST.MOCK or JAVASCRIPT projects using JEST.

Metric limitations. The metrics we used (generation, compilation, execution, and coverage) capture only part of test quality. Generated tests that compile and pass may still need improvement. Future work should include a manual assessment of the quality of generated tests.

Model bias and training data leakage. To mitigate data leakage, we selected projects updated after the training cut-off date of the selected models. Despite this precaution, it remains difficult to guarantee that the results stem purely from reasoning rather than memorization of data (possibly from similar projects) seen during training.

VI. RELATED WORK

Traditional Test Generation Techniques. Random testing [15] and search-based software testing [29] reduce manual effort by automatically producing tests to maximize structural coverage. Similar to MOCKMILL, these techniques leverage information in source code to enhance test generation. MSEQGEN [30] mines method call sequences from code repository to achieve higher coverage. Fraser and Zeller [31] mine object usage patterns from existing code to generate meaningful tests. Our work complements them by utilizing mocking information to guide test generation toward better tests.

Automated Test Generation with LLMs. LLM-based approaches generate unit tests via prompt engineering and structured context. Studies on zero-shot, few-shot, and Chain-of-Thought prompting show that richer, code-aware inputs (e.g., signatures, existing tests, stack traces) outperform plain natural language prompts [6]. Accordingly, pipelines typically collect project context, query an LLM, and iteratively compile/execute and repair failing outputs [8], [9], [21], [7]. Similar to prior work, MOCKMILL relies on prompt design and post-processing. However, unlike approaches that build context solely from source code and project metadata, MOCKMILL also leverages developer-written mocks and stubs as structured input to guide the LLM in generating realistic tests for mocked and stubbed classes. Although prior tools may generate tests with test doubles, none explicitly exploit existing mocking information to guide test generation. To the best of our knowledge, MOCKMILL is the first to do so.

Test Doubles in Software Testing. Recent work has introduced tools that analyze, generate, or refactor mocks and

stubs to improve testing quality and maintainability. MOCKSNIFFER [32] characterizes recommends mocking decisions. Empirical studies on mock assertions [14] suggest that mocks capture developers' behavioral intent and domain knowledge. MOCKMILL directly utilize this encoded intent to inform LLM-driven test generation. STUBCODER [33] generates and repairs stub code via evolutionary search to keep tests passing as production code evolves. RICK [34] records production executions and generates tests that mimic observed behavior using test doubles. ARUS [35] improves maintainability by detecting and removing unnecessary stubbings. Compared with these approaches, MOCKMILL does not aim to generate or improve test doubles but instead leverages the behavioral information already encoded in them to guide LLMs in generating new tests.

VII. Conclusions and Future Work

This paper introduced MOCKMILL, the first LLM-based test generation approach that leverages developer-written mocking information to guide test creation. MOCKMILL generates tests that uncover lines of code and mutants missed by baseline approaches, showing that mock information provides valuable contextual cues for producing realistic and useful tests. The approach complements developer-written suites, traditional tools such as RANDOOP, and an vanilla LLM-based approach. The cost overhead of 5–15% with respect to the LLM baseline considered is acceptable given the corresponding improvements in test quality and fault detection capability.

Our findings suggest that future LLM-driven test generation should incorporate available mocking information as contextual guidance to produce more comprehensive and realistic tests. Rather than replacing existing methods, mock-informed generation should complement them within a holistic LLM-based testing framework that integrates both mock-informed and non–mock-informed prompts.

As the first work of its kind, MOCKMILL opens several promising research directions. Future work includes ablation studies to assess the relative impact of mocking information, automatic extraction of project-specific few-shot examples to improve performance, dynamic collection of mocking data to capture richer behaviors, and establishing traceability links between test doubles and generated tests to better understand their influence on test creation.

References

[1] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[2] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *FOSE*. IEEE, 2007, pp. 85–103.

[3] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *JSS*, vol. 86, no. 8, pp. 1978–2001, 2013.

[4] G. Jahangirova and V. Terragni, "Sbft tool competition 2023 - java test case generation track," in *SBFT*, 2023, pp. 61–64.

[5] A. Bodicoat, G. Jahangirova, and V. Terragni, "Understanding llm-driven test oracle generation," in *ACM AIWARE*, 2025.

[6] W. C. Ouédraogo, K. Kaboré, H. Tian, Y. Song, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyandé, "Large-scale, independent and comprehensive study of the power of llms for test case generation," *arXiv*, 2024.

[7] V. Terragni, A. Vella, P. Roop, and K. Blincoe, "The future of ai-driven software engineering," *ACM TOSEM*, 2025.

[8] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE TSE*, vol. 50, no. 1, pp. 85–105, 2023.

[9] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE TSE*, vol. 50, no. 4, pp. 911–936, 2024.

[10] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[11] S. Mostafa and X. Wang, "An empirical study on the usage of mocking frameworks in software testing," in *QSIC*. IEEE, 2014, pp. 127–132.

[12] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "Mock objects for testing java systems: Why and how developers use them, and how they evolve," *ESEM*, vol. 24, pp. 1461–1498, 2019.

[13] M. Fazzini, C. Choi, J. M. Copia, G. Lee, Y. Kakehi, A. Gorla, and A. Orso, "Use of test doubles in android testing: An in-depth investigation," in *ICSE*, 2022, pp. 2266–2278.

[14] H. Zhu, V. Terragni, L. Wei, S.-C. Cheung, J. Wu, and Y. Liu, "Understanding and characterizing mock assertions in unit tests," *ACM PACSE*, vol. 2, no. FSE, pp. 554–575, 2025.

[15] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE*. IEEE, 2007, pp. 75–84.

[16] Lee, Jamie and Teh, Flynn and Zhu, Hengcheng and Li, Mengzhen and Fazzini, Mattia and Terragni, Valerio, "MockMill Replication Package," https://doi.org/10.5281/zenodo.19490389, 2026.

[17] Mockito Framework, "Mockito javadoc," https://javadoc.io/doc/org.mockito/mockito-core/latest/org.mockito/org/mockito/Mockito.html, 2025, accessed: March 2026.

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

[19] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[20] 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," *arXiv preprint arXiv:2501.12862*, 2025.

[21] R. Ravi, D. Bradshaw, S. Ruberto, G. Jahangirova, and V. Terragni, "Llmloop: Improving llm-generated code and tests through automated iterative feedback loops," *ICSME. IEEE*, 2025.

[22] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *ASE*. IEEE, 2015, pp. 201–211.

[23] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Evolutionary improvement of assertion oracles," in *FSE*, 2020, p. 1178–1189.

[24] JUnit Team, "Junit 5: The next generation of junit," https://junit.org/junit5/, 2025, accessed: March 2026.

[25] Mockito Framework, "Mockito: Tasty mocking framework for unit tests in java," https://site.mockito.org/, 2025, accessed: March 2026.

[26] V. Terragni, P. Salza, and M. Pezzè, "Measuring Software Testability Modulo Test Quality," in *ICPC*, 2020.

[27] PIT Mutation Testing, "Pit: State of the art mutation testing for java," https://pitest.org/, 2025, accessed: March 2026.

[28] JaCoCo Project, "Jacoco: Java code coverage library," https://www.jacoco.org/jacoco/, 2025, accessed: March 2026.

[29] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *FSE*, 2011, pp. 416–419.

[30] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Mseqgen: object-oriented unit-test generation via mining source code," in *FSE 2009*, 2009, pp. 193–202.

[31] G. Fraser and A. Zeller, "Exploiting common object usage in test case generation," in *ICST 2011*, 2011, pp. 80–89.

[32] H. Zhu, L. Wei, M. Wen, Y. Liu, S.-C. Cheung, Q. Sheng, and C. Zhou, "Mocksniffer: Characterizing and recommending mocking decisions for unit tests," in *ASE*, 2020, pp. 436–447.

[33] H. Zhu, L. Wei, V. Terragni, Y. Liu, S.-C. Cheung, J. Wu, Q. Sheng, B. Zhang, and L. Song, "Stubcoder: Automated generation and repair of stub code for mock objects," *ACM TOSEM*, vol. 33, no. 1, 2023.

[34] D. Tiwari, M. Monperrus, and B. Baudry, "Mimicking production behavior with generated mocks," *IEEE TSE*, 2024.

[35] M. Li and M. Fazzini, "Automatically removing unnecessary stubbings from test suites," in *ICST*. IEEE, 2024, pp. 233–244.