LLMs for Automated Unit Test Generation and Assessment in Java: The AGONETEST Framework

Andrea Lops*‡, Fedelucio Narducci*, Azzurra Ragone†, Michelantonio Trizio‡, Claudio Bartolini‡
*Polytechnic University of Bari, Bari, Italy
Email: {andrea.lops, fedelucio.narducci}@poliba.it
†University of Bari, Bari, Italy
Email: azzurra.ragone@uniba.it
‡Wideverse, Bari, Italy
Email: {andrea.lops, michelantonio.trizio, claudio.bartolini.consultant}@wideverse.com

***Abstract*—Unit testing is an essential but resource-intensive step in software development, ensuring individual code units function correctly. This paper introduces AGONETEST, an automated evaluation framework for Large Language Model-generated (LLM) unit tests in Java. AGONETEST does not aim to propose a novel test generation algorithm; rather, it supports researchers and developers in comparing different LLMs and prompting strategies through a standardized end-to-end evaluation pipeline under realistic conditions. We introduce the CLASSES2TEST dataset, which maps Java classes under test to their corresponding test classes, and a framework that integrates advanced evaluation metrics, such as mutation score and test smells, for a comprehensive assessment. Experimental results show that, for the subset of tests that compile, LLM-generated tests can match or exceed human-written tests in terms of coverage and defect detection. Our findings also demonstrate that enhanced prompting strategies contribute to test quality. AGONETEST clarifies the potential of LLMs in software testing and offers insights for future improvements in model design, prompt engineering, and testing practices.**

***Index Terms*—Software Testing, Large Language Model, Automatic Assessment and Evaluation, Assessment and Evaluation in Software Testing**

I. INTRODUCTION

Software testing is a critical step in the software development lifecycle, essential for ensuring code correctness and reliability. Unit testing, in particular, verifies the proper functioning of individual code units. However, designing and building unit tests is a costly and labor-intensive process that requires significant time and specialized skills [1]. Automating this process is an active area of research and development.

Automated tools for generating unit tests can reduce the workload of test engineers and software developers. These tools typically use static code analysis methods to generate test suites. For example, EvoSuite [2], a popular tool that combines static code analysis with evolutionary search, has demonstrated the ability to achieve adequate coverage.

Large Language Models (LLMs), efficiently exploited in various aspects of software development, could also handle the automatic generation of unit tests. Several empirical studies on LLMs have highlighted their ability to generate tests for simple scenarios, often limited to single methods [3]–[6]. Though directionally useful, these explorations often focus on independent, small-scale test units and rely on manual integration into projects, providing a limited view of LLM performance in real-world software development scenarios [6], [7]. This manual process restricts the number of tests that can be executed and reduces overall efficiency.

To address these gaps, we have developed a framework explicitly focused on the evaluation of unit test suites generated by LLMs. Rather than proposing a novel generation method, our contribution lies in providing an end-to-end pipeline that standardizes how LLM-based test suites can be assessed in realistic software projects. A simple use case illustrates how AGONETEST can be applied in a real-world scenario. Imagine a developer or a researcher who needs to evaluate which LLM and prompting strategy performs best for generating unit tests. Doing this manually would require repeated project setup, test execution, and metric collection, making the process slow and error-prone. With AGONETEST's standardized end-to-end pipeline, the developer can automate the workflow and directly compare LLMs under different prompting strategies. The framework produces reliable and reproducible metrics, revealing, for instance, that one model generates more compilable tests while another achieves higher coverage. In this way, AGONETEST turns ad hoc experimentation into systematic benchmarking. Our approach focuses on class-level test code evaluation, which is closer to real-world practices as it covers method interactions and shared state, reducing code redundancy [8].

For instance, a simple `ItemManager` class with two methods: `addItem()` and `getItemCount()`, illustrates this point. A method-level test for `getItemCount()` in isolation might only verify its behavior on an empty list, ignoring how the state changes when new items are added. In contrast, a class-level test naturally exercises the interaction between `addItem()` and `getItemCount()`, ensuring that the internal state is updated consistently across method calls. This example highlights three key benefits of class-level testing:

- **Reduction of redundancy**: common setup code (e.g., creating and initializing an object) can be reused across multiple methods.
- **Coverage of complex interactions**: tests verify how

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00198

methods behave together, capturing issues that method-level tests may miss.
- **Holistic view of class behavior**: class-level tests reveal whether the class as a whole fulfills its intended responsibilities, beyond the correctness of individual methods.

In this work, we introduce AGONETEST, an automated evaluation system for LLM-generated unit tests. AGONETEST integrates project setup, context extraction, execution of generated tests, and quality measurement using standard metrics. Leveraging the METHODS2TEST dataset [9], we developed a new dataset specifically aimed at comparing human-written tests with those produced by LLMs.

The **main contributions** of our work are as follows:

- **AGONETEST**[1]: we designed and developed a closed-loop, highly automated software system supporting the process of assessing LLM-generated unit tests, with automated project setup, prompt integration, and metrics computation.
- A unified framework for comprehensive evaluation of a variety of LLMs and relative prompting types and prompt schemata in the task of developing unit tests, and a set of metrics and test smells to assess the quality of the generated test suites;
- **CLASSES2TEST**[1]: An annotated open source Java project dataset extending METHODS2TEST [9], which maps classes under test to their related test classes. This extended dataset makes it possible to assess the test performance of an LLM on a more complex scope (the entire class) than the single method.

The paper is organized as follows. Section II sets the background and highlights differences between our work and related work. Section III gives an overview of AGONETEST and its modules, detailing their functional scope. Then, Section IV showcases how AGONETEST is applied in practice through an end-to-end example. Section V presents the Research Questions (RQs) guiding the experiment and the evaluation settings, while Section VI answers the research questions and discusses the results. Section VII discusses the limitations of our approach, and Section VIII concludes the paper, outlining potential directions for future work.

II. BACKGROUND AND RELATED WORK

A. Unit Test Generation

Unit test generation is the automated process of creating test cases for individual software components, such as functions, methods, or modules. These test cases are used to independently verify the correct functioning of each unit.

Present techniques employ randomness-based [10], [11], constraint-based [12], [13], or search-based approaches [14], [15]. The core idea behind these methods is to transform the problem into one that can be solved mathematically. For example, search-based techniques convert testing into an optimization problem, to generate unit test cases [16]. Consequently, the objective of these techniques is to generate all potential solutions and then select those that achieve better code coverage.

Tools like JUGE have been proposed to evaluate and compare Java unit test generators comprehensively. JUGE provides an infrastructure for benchmarking various Java unit test generation tools by measuring their performance across standard datasets and metrics, enabling a fair and systematic evaluation [17]. While JUGE focuses on benchmarking and comparing classical tools, AGONETEST not only allows unit test generation using multiple tools and techniques, including LLM-based approaches, but also integrates automated assessment into a single, unified framework, making it a practical solution for real-world software development workflows.

EvoSuite [2] generates unit tests for Java by applying search-based algorithms that evolve candidate test suites toward coverage objectives such as line and branch coverage. The tool evaluates test fitness iteratively through variation, selection, and optimization, and it automatically produces JUnit test cases along with detailed reports on metrics like code coverage and mutation score. Despite its popularity, EvoSuite has notable limitations. It often produces tests that lack clarity and readability [18], which hinders their practical usefulness. Moreover, EvoSuite only supports Java 9 or earlier versions and appears to be no longer actively maintained, restricting its applicability to modern Java projects. In contrast, AGONETEST overcomes this barrier by supporting all Java LTS versions.

Randoop [11] is a feedback-directed random test generator for Java programs. It works by repeatedly selecting sequences of method and constructor invocations, executing them, and using the observed program behavior to guide further generation. Randoop is lightweight and can quickly produce large numbers of test cases that expose common programming errors such as null dereferences, assertion violations, or unhandled exceptions [19]. However, the generated tests often require additional curation: they may include redundant or uninformative assertions, depend on specific runtime states, or fail to integrate smoothly into existing build systems. Unlike AGONETEST, which emphasizes project-level integration and automated assessment, Randoop is typically applied at the level of individual classes with per-class configuration, making large-scale, unattended evaluation less practical.

B. Large Language Models for Test Generation

Since the emergence of LLMs, they have been used for test suite generation. The first techniques exploiting LLMs were thought of as solutions to neural machine translation problems [20], [21]. Such approaches work by translating from the primary method to the appropriate test prefix or test assertion while also fine-tuning the LLMs using the test generation dataset. For instance, AthenaTest [21] optimizes BART [22] using a test generation dataset in which the source is the primary method along with its corresponding code context, and the result is the complete test case.

[1]https://anonymous.4open.science/r/classes2test

AthenaTest focuses mainly on generating method-level tests by fine-tuning a single model, while AGONETEST shifts the focus to the generation of class-level tests. Our approach makes it possible to use up-to-date LLMs and not constrain prompt design (our prompts can be customized), thereby handling more complex, real-world scenarios.

In light of the rapid evolution of instruction-tuned LLMs, the proliferation of methods for generating tests is on the rise, exploiting guided LLMs through appropriate prompts, as opposed to model fine-tuning [23], [24]. Several proposals for evaluating LLMs in test suite generation have emerged. For example, CHATTESTER [6] proposes a tool for evaluating and improving LLM-generated tests based on ChatGPT.

ChatTester focuses on improving and evaluating tests generated by a specific LLM (ChatGPT), but requires human intervention to evaluate the generated code and does not provide an evaluation of class-level tests on multiple LLMs. AGONETEST provides support instead for a variety of LLMs and evaluates each LLM's performance on a wide range of real-life Java projects. TESTPILOT [4] is also focused on generating and improving tests using LLMs on JavaScript code. Although TestPilot performs an automated evaluation, it lacks wider applicability to projects other than the 25 repositories it considers in the work provided as reference here. AGONETEST offers far broader applicability by using a dataset of 9,410 GitHub repositories, and automatically integrating test libraries into them. CEDAR [25] instead proposes a prompt construction strategy based on *few-shot* learning [26] and the Codex model[2] to generate tests.

Cedar uses a specific prompt construction strategy but does not incorporate a structured mechanism to evaluate multiple LLMs and prompt types in a unified framework. AGONETEST provides this by allowing the integration and evaluation of various prompt engineering techniques and LLMs, offering a more holistic approach to test generation. Guilherme and Vincenzi [3] use `gpt-3.5-turbo` in analyzing the impact of variation in model hyperparameters.

The study by Guilherme and Vincenzi presents an initial assessment but lacks automation in evaluating comprehensive test quality metrics like mutation score and test smells. AGONETEST goes a step further by automating these evaluations, integrating advanced metrics to provide a deeper analysis of the generated tests. Siddiq et al. [5] offer a new proposal for evaluating tests generated using common datasets and experimenting with the use of new metrics [27].

Although Siddiq et al. use Test Correctness (but not mutation score) on top of all the metrics that AGONETEST uses, their approach does not fully automate the test generation-execution-evaluation loop or focus on class-level tests. AGONETEST fills this gap by providing end-to-end automation and focusing on generating and evaluating complex, class-level test suites.

A number of other LLM-based test generation tools have been proposed, but they suffer from limitations that make them unsuitable for systematic comparisons with AGONETEST. For instance, A3TEST [28], CHATUNITEST [29], TESTSPARK [30], and PROJECTTEST [31] proved unusable in our experiments due to critical issues such as non-functional setup scripts, token limit errors even on small inputs, inability to generate a single compilable test without extensive manual intervention, and disregard for project-specific dependencies. Similarly, our analysis of TESTBENCH [32] revealed that its provided codebase was not functional out-of-the-box, producing immediate compilation errors and lacking the necessary requirements for systematic evaluation.

Other tools differ in goals or scope: for example, TESTGEN-LLM [33] requires manual validation for each test, thus breaking the automated loop that is central to AGONETEST. MUTAP [34] uses mutation score to improve prompt engineering, whereas AGONETEST employs mutation score as a final quality metric for assessing the robustness of the entire generated test suite. Finally, tools such as QODO COVER[3] and UTBOT[4] focus primarily on the generation process and have their target LLMs hard-coded, preventing comparative studies across multiple models and prompting strategies.

C. Limits of Current Approaches in Applying LLMs to Unit Test Generation

While promising, current approaches in applying LLMs to unit test generation exhibit several limitations:

a) Limited Scope: Current methods for assessing how useful LLMs are in test code generation are mostly limited to the generation of code segments, rather than whole modules or components (e.g., whole classes in Java). Consequently, until very recently, the research community lacked mature and widely adopted datasets for evaluating class-level test generation. Some early benchmarks have now started to appear, including TESTGENEVAL [35], TESTBENCH, and PROJECTTEST. These works confirm the importance of class- and project-level testing, but they differ from our contribution in scope and maturity: TESTGENEVAL focuses on single classes and limited metrics, TESTBENCH provides a prototype implementation with limited usability, and PROJECTTEST explores project-level mappings but lacks full automation.

Other studies often provide only punctual and anecdotal evaluations of the generated results, evaluating LLMs in the task of generating tests only at the method level or in contexts limited to sections of code [4], [6], [21].

b) Lack of Automation: While some prior works (e.g., TESTGENEVAL, TESTBENCH, PROJECTTEST) have started exploring aspects of automated evaluation, none provide a fully automated end-to-end framework that integrates project setup, test generation, execution, and metric computation in a reproducible pipeline. AGONETEST fills this gap by offering an extensible evaluation framework rather than a one-off benchmark.

[2]https://openai.com/index/openai-codex/

[3]https://github.com/qodo-ai/qodo-cover

[4]https://github.com/UnitTestBot

c) Subjective Choice of Prompts: In most cases, the choice of prompts to get LLMs to generate testing code remains subjective. There is no thorough evaluation of alternate prompting techniques compared to those initially proposed, leaving room for further exploration and optimization in prompt engineering. [5], [24], [25].

Table I summarizes the characteristics of these works and compares them with ours.

d) Contribution of AGONETEST*:* In contrast to these limitations, AGONETEST provides three distinctive features that set it apart from prior work. First, it offers full automation, covering the entire loop from repository setup to prompt instantiation, test generation, execution, and quality assessment, without requiring manual steps. Second, it explicitly targets class-level testing, which more closely reflects real-world development practices by capturing interactions between methods and shared state within classes, going beyond the common focus on isolated methods. Finally, AGONETEST is designed as an extensible framework, allowing researchers and practitioners to integrate new LLMs, prompt strategies, and datasets with minimal effort, thus serving as a reusable infrastructure rather than a one-off experiment.

III. OVERVIEW OF AGONETEST

The term *agone*, originating from ancient Greece and Rome, signified a contest wherein philosophers debated their ideas, with the audience determining the victor. We adopt the term *agone* metaphorically to represent the competitive evaluation of LLMs and their respective prompting strategies within an arena aimed at generating optimal unit test suites. AGONETEST determines the optimal strategies based on standard test quality metrics, see Sec. III-E.

AGONETEST is designed to provide software testers with a system for generating and assessing unit tests. This assessment focuses on key metrics such as code coverage, defect detection rate, and the presence of known test smells, thereby offering a comprehensive assessment of test suite quality.

AGONETEST operates on the principle that the evaluation of LLMs in the task of generating high-quality unit tests can be performed through the collaboration of test engineers and data scientists (or prompt engineers). However, in practice, a single experienced test engineer familiar with generative AI can perform both roles, allowing the focus to be only on defining new prompt types and the comparison of LLMs. This is the persona that we evoke when we refer to the AGONETEST user (alternatively, "the test engineer") in the remainder of this paper.

Figure 1 provides a high-level diagram of the architecture of AGONETEST, showing the operating modules that streamline the test generation and evaluation process. The framework can be described as follows:

1) Strategy Configuration:

a) Sample Projects Selection: As an initial configuration step, the user chooses which repositories to generate test suites for. This initial phase leverages a comprehensive dataset of open-source Java repositories, which we contribute to the community.

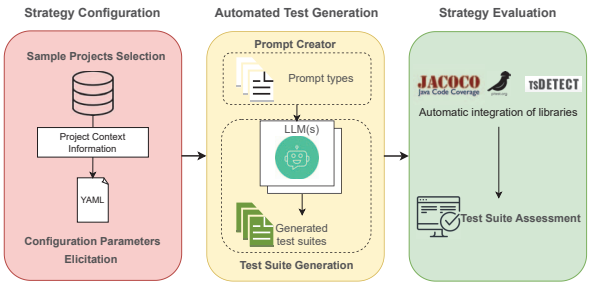

Fig. 1. Overview of AGONETEST framework

b) Configuration Parameters Elicitation: In this phase, configuration parameters are elicited from the selected repositories (e.g., the project Java version, the used testing framework, etc.) and processed to create prompt templates that will be handed over to the LLMs.

2) Automated Test Generation:

a) Prompt Creator: During this phase, the prompt templates built in the previous phase are fully instantiated and then used to generate unit test suites in the next step.

b) Test Suite Generation: Here, AGONETEST orchestrates the interaction with the user's selected LLMs, feeding them the instantiated prompts to produce the unit test code. Each LLM generates test classes, which are then automatically integrated into the project structure.

3) Strategy Evaluation:

a) Test Suite Assessment: This phase assesses the quality of the test suites by computing various metrics and identifying (if any) test smells. This assessment enables a detailed analysis of the effectiveness and quality of the automated test generation strategies.

In the following, we describe each phase of the process in detail.

A. Sample Projects Selection: CLASSES2TEST dataset

We note that the AGONETEST tool can import and process any Java industrial project. For the purpose of evaluating our system, we built a comprehensive and annotated dataset of open-source Java repositories from GitHub, which we leverage in this phase. Unlike popular datasets in the literature, our dataset enables the generation and validation of unit tests at the Java class level, rather than at the individual method level. To collate and annotate our dataset, we built on METHODS2TEST [9]. The METHODS2TEST dataset was originally built from an initial pool of 91,385 GitHub repositories. To ensure realism and maintainability, only projects satisfying strict criteria were retained, yielding approximately 10% of the initial corpus (9,410 repositories). The filtering procedure excluded repositories that:

- "had not been updated in the past five years,"
- "were forks or duplicates,"

TABLE I
COMPARISON BETWEEN AGONETEST AND RECENT CLASS-/PROJECT-LEVEL TEST–GENERATION BENCHMARKS/FRAMEWORKS.

Work	Language(s)	Input scope	Primary aim	End-to-end automation (setup→metrics)	LLM/ prompt configurable	Reported metrics	Build/ deps integration	Post-gen repair/ compile boost
AGONETEST (this work)	Java	Class-level within full project	Evaluation framework (multi-LLM, multi-prompt)	Yes	Yes	Line, Branch, Method Coverage Mutation score, 18 Test smells	Yes (Maven/Gradle)	Yes (enhanced prompting for imports/paths)
TESTGENEVAL [35]	Primarily Python	Single class (not full project)	Benchmark of generation quality	No	No	Line Coverage, Mutation score	N/A (no project build)	No
TESTBENCH [32]	Java	Class-level (repo subset)	Benchmark/analysis of LLM tests	No	No	Line, Branch, Method Coverage, Mutation score	No	No
PROJECTTEST [31]	Java	Project-level mapping focus	Project-level benchmarking	No	No	Line, Branch, Method Coverage, Mutation Score	No	No

- "failed to compile successfully after dependency resolution,"
- "or did not contain a sufficient number of paired method–test mappings."

This process ensured that the resulting dataset captures actively maintained, compilable, and representative real-world projects. For CLASSES2TEST, we extracted information not only on classes and their mapped test classes under test but also on test frameworks and Java versions used across projects: JUnit 4 (55%), JUnit 5 (41%), and others (4%, including TestNG). Regarding Java versions, the majority of repositories target Java 11 (42%), followed by Java 17 (25%), newer LTS versions such as Java 21 (18%), and Java 8 (14%). We chose METHODS2TEST as the starting point, as it contains not only the test methods to be tested, but also the corresponding test methods written and validated by humans. Human-written tests are a valid evaluation benchmark to evaluate the effectiveness of different LLMs in building a test suite.

To build the CLASSES2TEST dataset, we extracted all references to the open-source repositories present in METHODS2TEST in order to map the Java classes, referred to as *classes under test*, to their corresponding test classes.

Here is the process we followed to create CLASSES2TEST:

1) Extract the repository reference, GitHub URL, and selected branch;
2) Select the classes considered in METHODS2TEST;
3) Clone the repository and save the commit hash;
4) Map and save the classes under test along with their respective test classes.

To map classes under test to their corresponding test classes in CLASSES2TEST, we adopted a conservative two-step procedure. First, candidate test classes are identified by common Java naming conventions (e.g., `MyClassTest`, `TestMyClass`) within the `src/test/java` folder, also considering mirrored package structures. Second, these candidates are statically validated through AST analysis to confirm that they actually exercise the class under test, checking for imports, constructor calls, method invocations, or mocking references. Only pairs with sufficient structural evidence are retained. In cases where a test class references multiple classes under test, we compute an evidence ratio and keep the mapping only if one class under test dominates ($\geq$60%of references); otherwise, the test class is discarded. Ambiguous or conflicting mappings (e.g., two classes under test pointing to the same test class with comparable evidence) are excluded to ensure precision. In the rare cases where a candidate test class exhibited comparable structural evidence for multiple classes under test (i.e., no dominant mapping could be established), we conservatively excluded the mapping altogether to avoid spurious links. This situation occurred only in 3 instances out of the entire corpus ($\approx$0.002% of all analyzed classes), and the affected pairs were removed from CLASSES2TEST. This process was inspired by the methodology used in METHODS2TEST.

TABLE II
CLASSES2TEST DATASET CHARACTERISTICS

Characteristic	Value
# Test Classes	147,473
# Unique Repositories	9,410
Average Lines of Code per Class	1,178 (IQR: 420–1,960)
Average Cyclomatic Complexity per Class	55.3 (IQR: 18–92)
Test Framework Distribution	JUnit 4 (55%), JUnit 5 (41%), Other (4%)
Java Version Distribution	8 (14%), 11 (42%), 17 (25%), 21+ (18%)

The resulting dataset contains 147,473 test classes extracted from 9,410 unique repositories. A summary of the dataset's characteristics is shown in Table II.

B. Configuration Parameters Elicitation

Before unit test generation can begin, the system extracts some parameters from the projects selected in the previous step. These parameters are then fed into the module that instantiates prompts and selects LLMs. To query the model under examination, various prompting techniques are available and can be chosen by the test engineer [36].

The configuration parameters include:

- `class_under_test`: This variable contains the Java class for which the test suite must be generated;
- `testing_framework`: This variable provides the name and version of the project's testing framework (e.g., JUnit 4), directly extracted from the project during execution;
- `java_version`: This variable allows you to retrieve the version of Java that the project uses.
- `example_class_under_test & example_test_class`: These variables contain an example class under test and the corresponding test class extracted from a reference repository, useful to provide an example to the LLM if one wants to use the few-shot prompting technique;

- `example_testing_framework & example_java_version`: These variables provide the information about the example repo. The example consists of a class under test and a test class extracted from an open sample repository[5]. Future iterations of AGONETEST will automate exemplar selection to reduce bias and better reflect real-world scenarios.

See Sec. IV-A for an example of a real implementation.

C. Prompt Creation

In this phase, the prompt templates described in the previous phases are fully instantiated to create viable prompts to guide the LLM in generating unit tests. We populate the user-supplied prompt types by replacing the variables outlined in Sec. III-B.

It has to be noted that, in order to make sure our experiments and findings are reproducible, we prepared CLASSES2TEST by saving the commit hashes of the repositories used as sources. This allows AGONETEST to consistently extract information such as the Java version used, the type of test framework (e.g., JUnit), and its version.

Unlike previous approaches to creating unit testing with LLMs that require human intervention to input code information [3], [6], AGONETEST automates the process to a far greater degree. AGONETEST employs ElementTree [37] and a parser to read and modify the Maven and Gradle build (see Sec. III-E). It analyzes the libraries present and the Java version used in each build system. The prompting engine of AGONETEST is designed to be extensible and flexible. Beyond the zero-shot and few-shot strategies illustrated in our examples (see Sec. IV-A), the YAML-based configuration supports custom variables that users can freely define to adapt prompts to new scenarios. This design makes it straightforward to incorporate advanced prompting techniques such as multi-shot prompting (with multiple test pair examples) or retrieval-augmented generation [38] (injecting external context). The engine does not hard-code prompt types, but instead provides a generic schema that researchers and practitioners can extend for novel experimental setups.

D. Test Suite Generation

At this point in the process, we have everything we need for the selected LLMs to generate test suites for each class under test of the project. To ensure each model has an appropriate number of tokens, we use tiktoken[6], a BPE tokenizer [39], to evaluate the token count in the prompt. When the token limit of a target model is exceeded, AGONETEST provides configurable fallback strategies rather than failing silently. By default, the system notifies the user of the excess and the number of tokens required. The user can then: (i) truncate the input, for instance by shortening comments or omitting less relevant methods; (ii) manually adjust the prompt template to reduce verbosity or context.

We remark that AGONETEST allows users to automatically choose and evaluate a wide range of LLMs. This capability is provided by the open-source LiteLLM library[7], which facilitates communication with more than 100 models[8] using a standard interaction based on the OpenAI API format[9]. Integration is made easier by LiteLLM, which translates inputs to satisfy the unique endpoint needs of each provider. This is crucial in today's environment, where the absence of standard API specifications for LLM providers makes it challenging to incorporate several LLMs into projects.

After invoking the LLM, AGONETEST selects relevant information from the LLM's answer (i.e., the generated test class). This step is crucial for automating the entire process, since LLMs can provide detailed descriptions or explain how the code should be structured without actually generating it [29]. In this component, AGONETEST removes unnecessary parts (like outline descriptions) and creates a new file to integrate the test class into the project.

E. Test Suite Assessment

Here, we evaluate the quality of the test suite according to the quality metrics described in the following. Unlike previous methods [5], [6] that require manual or partial automation, our framework provides a fully automated evaluation of the generated tests through systematic integration of evaluation metrics, representing a distinct step forward in automated evaluation. It is important to note that this component is separate from the experimental evaluation discussed later. Instead, it serves as an additional tool provided by AGONETEST to assist engineers in assessing the quality of the generated tests.

1) Code Coverage: we calculate several coverage metrics, specifically: Line coverage, Method coverage, and Branch coverage [40]. To measure code coverage in test suite execution, we integrated in AGONETEST the JaCoCo[10] library.

2) Defect detection rate: to measure the robustness of the test suite, we use mutation score. The mutation score evaluates the capability of tests to detect syntactic changes (mutants) artificially introduced in the code. This provides a widely used and reliable proxy for test suite robustness, as shown in prior literature [41].

3) Test Smells [27]: we decided to identify test smells in the generated test suite as proxy indicators of potential quality and maintainability issues in the test code. Although test smells do not directly measure functional correctness, their presence often correlates with problematic patterns that could negatively affect the effectiveness and readability of the generated tests. To identify these smells in the generated test suite, we integrate the library TSDETECT [42]. AGONETEST determines whether the following test smells are present in the code: Assertion Roulette (AR) [43]; Conditional Test Logic (CTL) [44]; Constructor Initialization (CI) [45]; Default

[5]https://github.com/junit-team/junit5-samples

[6]https://github.com/openai/tiktoken

[7]https://github.com/BerriAI/litellm

[8]https://docs.litellm.ai/docs/providers

[9]https://platform.openai.com/docs/guides/text-generation/chat-completions-api

[10]https://www.jacoco.org/jacoco/index.html

Test (DT); Duplicate Assert (DA) [45]; Eager Test (EA) [43]; Empty Test (EM) [45]; Exception Handling (EH) [45]; General Fixture (GF); Ignored Test (IT) [45]; Lazy Test (LT) [43]; Magic Number Test (MNT) [44]; Mystery Guest (MG); Redundant Print (RP) [45]; Redundant Assertion (RA) [45]; Resource Optimism (RO) [45]; Sensitive Equality (SE) [43]; Sleepy Test; Unknown Test (UT) [45].

After adding the necessary libraries, AGONETEST runs a build and test to ensure that there are no compilation errors. The assessment phase of the test suite of our process presents a high degree of automation, as we describe below.

In this phase, AGONETEST automatically includes these libraries into the project. For each run, AGONETEST checks the configuration files of the supported build systems (Maven and Gradle, Sec. III-C) to determine if the necessary libraries are already present. If they are not, it modifies the configuration to add the required dependencies.

AGONETEST provides a high degree of automation, for example, in its handling of the PiTest library. Specifically, if the repo uses the JUnit 5 test framework, an additional library, "pitest-junit5-plugin", is required. Utilizing information extracted from the repo in the Prompt Creation module (Sec. III-C), AGONETEST automatically identifies the test framework in use and adds this dependency without any human intervention.

AGONETEST generates a report with the results of the quality metrics computed for the LLM-generated test suite. To achieve this, the tool automatically retrieves detailed information from the reports produced by the libraries, compiling this data for each class within each project.

Failure handling.: All coverage and mutation metrics are computed on the full evaluation set by assigning a value of 0 to non-compiling generations (Sec. V-A). Test smell counts are build-penalized in the same way (non-compiling $\rightarrow$ 0 contribution). We also report the compilation rate to contextualize the aggregated scores.

It is important to note that AGONETEST is not limited to the CLASSES2TEST dataset. Thanks to its modular architecture, the framework can seamlessly integrate with alternative benchmarks such as DEFECTS4J [46] or any other curated corpus of Java projects. Researchers need only adjust the configuration files to extend the evaluation to new projects.

IV. AGONETEST IN PRACTICE

In this section, we will demonstrate how AGONETEST operates in practice by describing an end-to-end run of a practical example.

We will skip the repository selection phase in our account and move straight to the configuration phase, which concerns LLM selection and prompt specification. Then, we will exemplify how the results are presented back to the user for further analysis.

A. Configuration

As described in Sec. III-B, AGONETEST utilizes a YAML[11] file as input, where it is possible to specify information related to two elements: `llms` and `prompts`. The YAML file represented in the Listing 1 is an example configuration for AGONETEST that will be used in Sec. IV.

Listing 1. Setup of the YAML configuration file: setting of variables for two different LLMs and two different prompts.

```
llms:
- model: gpt-4o-mini
  temperature: 0
- model: gemini-1.5-pro
  temperature: 0
- model: llama3.1:70b
  temperature: 0
prompts:
- name: zero-shot
  value:
  - role: system
    content: You are provided with Java class. Create a test
          class that fully tests the proposed Java class
         using the project information for imports. Reply
         with code only, do not add other text that is not
         code.
  - role: user
    content: "The project uses {{testing_framework}} and
         Java {{java_version}} and Java class is:  \n<code>\
         n {{class_under_test}}\n</code>"
- name: few-shot
  value:
  - role: system
    content: You are provided with an example with a Java
         class and its test class. You are then provided
         with a new Java class. Take a cue from the example
         and create a test class that fully tests the new
         proposed Java class. Reply with code only, do not
         add other text that is not code.
  - role: user
    content: "#Example:\nThe example Java class is:\n<code>\
         n {{example_java_class}} \n</code>\nThe example
         test class is: \n<code>\n {{example_test_class}} \n
         </code>.\nThe Java class you must create the test
         for is: \n<code>\n{{class_under_test}}\n</code>"
```

The LLM is specified by setting the model name in the `model` field, selecting from the available models supported by LiteLLM[12]. In the `temperature` field, you can set the temperature at which the model will operate. The temperature can have a value between 0 and 2. Higher values (e.g., 2) produce more random outputs, while lower values (e.g., 0) make the outputs more targeted and deterministic [3].

The `prompts` field consists of two sections: `name` and `value`. The former is an identifier for labeling the type of prompt (*zero-shot*, *few-shot*, etc.); the latter, `value`, is an array of message elements of type OpenAI[13]. Each individual message includes a `role` and a `content`.

For the `role` field, there are two possible values:

- `system`: used to instruct the model on the behavior it should adopt.
- `user`: used to indicate the request for the generation of the test class.

Two types of prompts can be specified:

- `zero-shot`: refers to presenting the model with a single instance of a request or task without any previous

[11]https://yaml.org/

[12]https://docs.litellm.ai/docs/providers

[13]https://platform.openai.com/docs/api-reference/chat/create#chat-create-messages

examples for the model to draw upon [47]. This method emphasizes the model's ability to comprehend and accurately execute the given task.
- `few-shot`: unlike zero-shot prompting, few-shot prompting involves providing the model with examples demonstrating the expected inputs and outputs [26]. This technique aids in contextual learning by including examples in the prompt, thereby guiding the model towards improved performance. These examples serve as a conditioning for the actual request, helping the model generate more accurate and relevant responses.

This configuration file will instruct AGONETEST to perform the steps described in Sec. III-C: it will fully instantiate template variables, including class under test, test frameworks used by the repos (and versions thereof), and version of the JDK used in the repos.

B. Results presentation

After performing the generation step, AGONETEST produces a report containing, for each selected LLM and each prompting mode, the quality metrics for the classes under test and the entire project. Table III shows an excerpt of the report that also contains human-written test results as they were present in the CLASS2TEST dataset. The report provides valuable information on the strengths and weaknesses of each combination of LLM and prompt, showing the average value for each metric.

In this way, software testers can accurately assess the effectiveness of the LLM in creating usable and effective class-level tests.

V. EXPERIMENT SETUP

In this experimental evaluation, our aim is to address the following research questions:

- **RQ1: How do the chosen LLMs and prompt types perform in the test case generation task?** We assess this by computing the quality metrics defined in Sec. III-E.
- **RQ2: How frequently do compilation errors occur, and how do they impact the overall compilation success rate?** We study and classify the most common errors that impact the unit test generation process.
- **RQ3: Is there a strategy to increase the compilation success rate?** We study how to increase the compilation success rate. In particular, we define an enhanced strategy to improve the compilation success rate.

A. Evaluation protocol: compiled-only averaging (build failures excluded)

Let N be the number of classes under test (since we performed the experiments on the entire CLASSES2TEST dataset, the number of classes will be 147,473). For each class i, let $\mathrm{build}_i \in \{0,1\}$ indicate whether the generated tests compile, and let $m_i \in [0,100]$ denote a metric value (e.g., line, branch, method coverage, or mutation score) computed only if $\mathrm{build}_i = 1$. Define $N_{\text{comp}} = \sum_{i=1}^{N} \mathrm{build}_i$. We report compiled-only averages (no penalty for non-compiling generations):

$$\overline{m} \;=\; \frac{1}{N_{\text{comp}}} \sum_{i=1}^{N} \mathrm{build}_i \, m_i.$$

For test smells, let $s_{k,i} \geq 0$ be the count for smell k on class i (defined only if $\mathrm{build}_i = 1$). We report compiled-only averages:

$$\overline{s_k} \;=\; \frac{1}{N_{\text{comp}}} \sum_{i=1}^{N} \mathrm{build}_i \, s_{k,i}.$$

We additionally report the compilation rate $R_{\text{build}} = \frac{1}{N} \sum_{i=1}^{N} \mathrm{build}_i$ to make the influence of failures explicit.

B. Model and Prompt Selection

We focus on evaluating the performance of LLMs in generating test cases on different types of models. For this purpose, we select `gpt-4o-mini`, `gemini-1.5-pro`, and `llama3.1:70b` for our experimentation. `gpt-4o-mini` is selected because of its excellent performance in code generation tasks [48] and has a number of parameters comparable with the other selected models[14]; `gemini-1.5-pro` because it is a model that can handle a context of 2 million tokens[15] and allows us to understand whether there are differences in performance based on context size; `llama3.1:70b` is selected because it is the open-parameter model that has comparable performance with others in the code generation task[16]. `llama3.1:70b` is hosted through Ollama[17]and queried locally, while `gpt-4o-mini` and `gemini-1.5-pro` models are queried via API.

In addition to choosing LLM, it is crucial to set its temperature parameter. By carefully setting the temperature parameter, users can balance innovation and coherence in text generation, ensuring that the output aligns with their specific task or application requirements. As shown in Listing 1, we set the temperature to 0 in our experiment to increase the level of coherence in text generation (and to decrease the level of randomness) and make the different test suites generated reproducible; results are single-run per model/prompt, and AGONETEST exposes temperature for repeated-run studies. Regarding prompt types, we experimented with two of the most popular techniques: zero-shot and few-shot, introduced in Sec. IV-A.

VI. EVALUATION AND RESULTS

A. *RQ1*: How do the chosen LLMs and prompt types perform in the test case generation task?

We analyze each LLM/prompt combination under the evaluation protocol in Sec. V-A, i.e., full-set averaging with non-compiling generations contributing zeros. As Table IV shows,

[14]https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence

[15]https://developers.googleblog.com/en/new-features-for-the-gemini-api-and-google-ai-studio

[16]https://ai.meta.com/blog/meta-llama-3-1/

[17]https://ollama.com

TABLE III

EXCERPT OF THE REPORT PRODUCED BY AGONETEST, SHOWING RESULTS FOR CLASSES UNDER TEST AND SELECTED COMBINATIONS OF LLMS AND PROMPT TYPES. THE FULL REPORT IS AVAILABLE IN THE ONLINE APPENDIX. HERE WE DISPLAY ONLY A REPRESENTATIVE SAMPLE FOR ILLUSTRATION. TEST SMELLS ARE REPORTED FROM COLUMN 10 ONWARDS (SEE SEC. III-E FOR ACRONYMS).

model	prompt name	Project	Class under test	branch coverage	line coverage	method coverage	mutation score	AR	CTL	CI	DA	EA	EM	EH	IT	LT	MNT	RP	RA	RO	SE	UT
gpt-4o-mini	few-shot	145256500	Key	35.71	64.86	84.62	33.33	10	0	0	0	10	0	0	0	28	10	0	0	0	1	0
gemini-1.5-pro	few-shot	145256500	Key	57.14	81.08	85.62	33.33	10	0	0	0	0	0	0	0	9	9	5	0	0	0	0
human	-	145256500	Key	28.57	37.84	30.77	25.0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
llama3.1:70b	zero-shot	43246449	FileHandler	-	100	100	0	1	0	0	0	0	0	0	0	4	4	0	0	1	0	0
human	-	43246449	FileHandler	-	100	100	100	1	0	0	0	0	0	0	0	4	4	0	0	1	0	0
gpt-4o-mini	zero-shot	1341207	HexString	50	92.11	100	85.71	7	0	0	0	18	0	0	0	45	18	0	0	0	1	0
gpt-4o-mini	few-shot	1341207	HexString	48.39	89.47	96.0	85.71	7	0	0	0	17	0	0	0	38	17	0	0	0	1	0
human	-	1341207	HexString	25.81	44.74	44.0	51.43	1	0	0	0	1	0	1	0	0	1	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

TABLE IV

AVERAGE OF METRICS COMPUTED FOR EACH MODEL AND PROMPT TYPES USED AND FOR HUMAN-WRITTEN TESTS. IN BOLD, THE BEST RESULTS FOR EACH METRIC. ONLY THE TEST SMELLS WITH NON-ZERO VALUES ARE SHOWN IN THE TABLE.[18]

model	prompt name	branch coverage%	line coverage%	method coverage%	mutation score%	AR	EH	MG	EA	LT	UT	RO	MNT
gpt-4o-mini	zero-shot	41.9%↓	64.8%↓	77.2%↑	44.5%↑	1.28	0.79	0.19	2.11	4.52	0.33	0.11	7.05
	few-shot	62.1%↑	71.3%↑	81.1%↑	61.0%↑	1.01	0.36	0.08	1.54	3.18	0.26	0.08	6.60
gemini-1.5-pro	zero-shot	22.6%↓	55.4%↓	62.8%↓	30.1%↓	1.67	0.42	0.09	2.85	3.71	0.29	0.10	5.50
	few-shot	44.6%↓	**89.8%**↑	**92.9%**↑	72.0%↑	1.96	0.41	**0.03**	2.92	3.25	**0.03**	0.19	5.15
llama3.1:70b	zero-shot	71.2%↑	80.3%↑	84.9%↑	32.0%↓	1.77	**0.27**	0.07	**0.51**	**1.58**	0.49	0.12	6.20
	few-shot	**79.8%**↑	85.6%↑	90.3%↑	**89.2%**↑	**0.55**	0.30	0.05	1.31	1.72	0.10	**0.03**	**3.85**
human	-	48.7%	73.2%	74.0%	40.4%	2.18	0.69	0.10	1.68	3.05	0.58	0.08	4.22

model and prompt choices materially influence all quality metrics.

1) Coverage and mutation: The highest mutation score is achieved by `llama3.1:70b` with few-shot prompting (89.2%), which also leads to branch coverage (79.8%). The highest line and method coverages are obtained by `gemini-1.5-pro` with few-shot prompting (89.8% and 92.9%, respectively). `gpt-4o-mini` shows balanced results, with few-shot outperforming its zero-shot configuration across metrics.

2) Test Smells: Test smells are indicators of poor test code quality and maintainability. Table IV reports the most relevant smells. In general, LLM-generated tests are comparable to human-written ones for Assertion Roulette (AR), Mystery Guest (MG), and Resource Optimism (RO). However, they exhibit better Exception Handling (EH), particularly with `llama3.1:70b`. This model also shows fewer instances of Eager Test (EA) and Lazy Test (LT) in its zero-shot configuration. The Unknown Test (UT) smell is rare across all LLM-generated tests. Conversely, LLMs tend to use fewer hard-coded values (Magic Number Test - MNT), especially with few-shot prompts, with the exception of `gpt-4o-mini` (zero-shot).

3) Prompt Types Comparison: The choice between zero-shot and few-shot prompting has remarkable effects: few-shot prompts improved performance in most models compared to human-written tests, particularly in line and method coverage for `llama3.1:70b` and `gemini-1.5-pro`. This effect underscores that exposure to example inputs improves prompt generation for class-level testing tasks.

[18]All values are full-set averages with build failures contributing 0. Arrows indicate whether a model–prompt configuration performs better (↑) or worse (↓) than the human baseline for that metric. Bold indicates the best score across all configurations.

RQ1 Findings

- **Coverage and defect rate:** `llama3.1:70b` (few-shot) with 89.2% mutation score and 79.8% branch coverage.
- **Effect of context:** The `gemini-1.5-pro` model prompt performed well in line and method coverage when using few-shot prompts, highlighting the importance of providing contextual examples.
- **LLM vs human:** LLM-generated tests matched or exceeded human-written tests on some quality metrics, when considering only the subset of tests that compile.

B. *RQ2: How frequently do compilation errors occur, and how do they impact the overall compilation success rate?*

After the initial experimental run, we observed relatively low compilation success rates across all tested LLMs, limiting the effective coverage analysis to successfully compiled classes. Therefore, the coverage metrics reported reflect only this subset of valid tests (see Table V). To investigate this

TABLE V
PERCENTAGE OF BUILD SUCCESS ON THE FULL SET PER EXPERIMENT

model	prompt name	Build%
gpt-4o-mini	zero-shot	28.6%
	few-shot	25.3%
gemini-1.5-pro	zero-shot	18.6%
	few-shot	**36.0%**
llama3.1:70b	zero-shot	9.8%
	few-shot	7.1%
human	-	100%

issue, we systematically collected and analyzed errors that prevented a successful compilation during our experiments. Two developers worked to categorize the errors. Each developer received a list of errors along with the corresponding generated code that triggered these errors. They independently labeled the errors without consulting each other. Upon completing their independent labeling, they joined to compare and discuss the results. A reconciliation process was conducted to resolve discrepancies in their labels. If there are any mismatches, the two developers discuss the differences to reach a consensus. In cases where they could not agree, a third developer acted as an arbitrator to make the final decision. The analysis of the errors revealed several recurring patterns that impacted the compilation success rate. These errors can be grouped into three main categories:

1) Symbol and Reference Issues:

a) Cannot Find Symbol: This was the most frequent error, occurring in 42.50% of cases. It typically involves references to variables, methods, or classes that are not defined or incorrectly named. This highlights the challenges the LLM faced in accurately recalling or generating the necessary components of the code.

b) Missing or Incorrect Imports: Errors due to missing import statements or incorrect library versions account for 19.55% of the total errors. This issue often arose from a mismatch between the generated code and the actual project dependencies, highlighting limitations in the models' understanding of the project environment.

2) Code Structure and Consistency Issues:

a) Override/Implementation Issues: Errors related to incorrect overrides or failed implementation of interfaces contribute to 13.87% of the errors. These errors often occur when the generated methods do not match the expected signatures or fail to fulfill the requirements of implemented interfaces.

b) Visibility/Access Issues: These errors, making up 7.69%, are due to incorrect access modifiers or attempts to access private or protected members from outside their intended scope. This indicates that the model sometimes struggled with correctly applying Java's access control rules.

c) Incorrect Data Types: Errors also arise from mismatched data types, accounting for 6.31% of the cases. This typically happens when the generated code attempts to pass arguments of one type to methods expecting a different type, suggesting challenges in maintaining type consistency across the generated test suite.

d) Instantiation Issues: Errors involving incorrect object instantiation account for 3.66% of the total. These often come up when the generated code uses the wrong constructor or attempts to instantiate abstract classes.

3) Syntax and Specific Rule Violations:

a) Syntax Errors: These include incorrect method signatures, missing semicolons, and other syntactic issues, making up 5.17% of the errors. Such errors suggest that, while LLMs can generate logically coherent code, they occasionally fail to adhere to the strict syntactical rules of Java.

b) Final Variable Issues: Errors related to improper use of final variables represented 1.26% of the cases. These typically involve attempts to reassign values to final variables, reflecting misunderstandings in handling Java's immutability constraints. The `gemini-1.5-pro` model stands out for achieving the highest compilation rate among the models tested, reaching 36.0% in the few-shot configuration. This outcome highlights the need to increase the compilation rate further, which we address in RQ3.

RQ2 Findings

- The most common issues affecting the compilation success rate are related to missing symbols and incorrect references (62.05%).
- Code structure problems, including overrides, access control, and data types, contributed 31.53%.
- Syntax and specific rule violations, such as improper use of final variables, accounted for 6.43%.

C. *RQ3: Is there a strategy to increase the compilation success rate?*

After analyzing the distribution of errors identified in the generated test suites, we observed that a significant portion of compilation failures were attributed to missing symbols or incorrect references, categorized under symbols and references issues. To address the compilation rate issues, we implemented an enhanced prompting strategy, explicitly providing the full path of classes under test within prompts. This refinement significantly increased compilation success rates by enabling LLMs to generate more contextually accurate import statements and symbol references. The enhanced strategy involves expanding the set of configuration parameters provided to the LLM by adding a new parameter, called "`class_under_test_path`". This parameter explicitly specifies the path to the class under test within the project, providing a clearer reference to ensure that the model has access to accurate and correct imports and symbol definitions related to the class under test. In addition, the prompts were updated to include this parameter, which provided a more grounded understanding of the model with respect to the dependencies and structure of the class under test. A compact before→after summary is reported in Table VI.

TABLE VI
COMPACT PER-MODEL BEFORE→AFTER DELTAS. VALUES ARE ABSOLUTE PERCENTAGE-POINT CHANGES.

model	prompt	Δ Build (pp)	Δ Branch	Δ Line	Δ Method	Δ Mutation
gpt-4o-mini	zero-shot	+28.3	+3.9	+3.7	+2.8	+4.1
	few-shot	+26.8	+3.1	+3.4	+2.5	+3.2
gemini-1.5-pro	zero-shot	+6.2	+1.3	+1.9	+1.5	+1.7
	few-shot	+28.6	+2.0	+2.2	+1.8	+2.6
llama3.1:70b	zero-shot	+14.3	+1.8	+1.6	+1.2	+2.1
	few-shot	+13.8	+1.9	+2.0	+1.6	+2.7

These improvements demonstrate that providing more precise details during the test generation process can significantly increase the chances of generating compilable code. The focus on explicit class under test paths allowed the LLMs to better handle project-specific imports and references, thereby addressing one of the major limitations observed in our initial experiments.

RQ3 Findings

- The enhanced strategy consistently raises build rates for all models and yields small but systematic increases in coverage and mutation score.
- Explicitly specifying the class under test path provides a more accurate reference, reducing errors related to missing symbols and incorrect references.
- Despite these improvements, additional strategies (e.g., automated post-generation repair or advanced context-aware prompting) are still needed to approach higher compilation success rates and functional correctness.

VII. LIMITATIONS

Although AGONETEST presents an innovative framework for automating the generation and evaluation of unit test suites using LLMs, we acknowledged some limitations of its current first implementation and experimental results.

a) Compilation Success Rate: The low compilation success rates observed represent a limitation inherent in the LLM-based code generation process rather than the AGONETEST framework itself. These limitations stem from the LLMs' challenges in understanding complex contextual dependencies. Future work could investigate new techniques to improve compilation rates, potentially by extending AGONETEST to incorporate automated code-repair strategies.

b) Dataset and Generalization: For our evaluation, we relied on a newly created CLASSES2TEST dataset which includes Java projects. This makes our findings not immediately generalizable to different programming languages. Moreover, the repositories included in CLASSES2TEST were selected based on their ability to compile without errors, potentially introducing a bias towards well-structured codebases.

c) Quality Metrics: Coverage, mutation score, and test smells are proxies of test effectiveness; they do not fully capture functional correctness.

d) Data Leakage: Data leakage is a concern when evaluating LLMs. We mitigate this by using a recently created dataset with projects updated after the training cutoff dates of the models used. However, this could become a more significant problem as models evolve.

e) Prompting: Our goal is an automated evaluation framework for LLM-based unit test generation; current experiments primarily illustrate how AGONETEST functions. The framework allows selecting LLMs and specifying prompts, but we explored only two prompting strategies, limiting full insight into model capabilities. Additionally, manual exemplar selection in one-shot prompts introduces subjectivity, raising concerns about prompt sensitivity and potential overfitting rather than genuine generalization.

f) Randomness/repeated runs: Results are from a single run with temperature=0; repeated-run variance analysis is left to future work and is supported by AGONETEST's temperature setting.

VIII. CONCLUSION AND FUTURE WORK

We introduced AGONETEST, an automated framework for evaluating unit test suites generated by LLMs on real-world Java projects. The main contribution of AGONETEST lies in its ability to provide a reproducible and extensible evaluation pipeline, integrating project setup, execution of generated tests, and quality metrics computation. To support this evaluation, we introduced the CLASSES2TEST dataset, which enables systematic benchmarking at the class level and complements existing resources such as METHODS2TEST. By focusing on evaluation rather than generation, AGONETEST offers the community a tool to compare LLMs, prompting strategies, and future improvements in test automation. As illustrated by our motivating use case, it transforms the ad hoc, error-prone process of manual evaluation into a systematic and reproducible workflow for developers and researchers. Our findings indicate that LLM-generated tests achieve comparable or superior code coverage and defect detection compared to human-written tests, particularly when context-aware prompting is used. This highlights the potential of LLMs in automating software testing, despite existing limitations. Future work will enhance AGONETEST by supporting more languages and boosting compilation rates. We also plan to expand the experimental scope with a wider variety of LLMs and advanced prompting techniques. Such enhancements will further empower developers, as described in our use case, to not only benchmark models but also to select and refine the optimal test generation strategy for their specific projects, thereby improving overall software quality.

ACKNOWLEDGMENT

This work was funded by: the European Union – NextGenerationEU under the *P+ARTS – Partnership for Artistic Research in Technology and Sustainability* (NRRP – M4C1, Investment 3.4, INTAFAM00037; CUP: G43C24000640006), *PE9 GRINS – Growing Resilient, INclusive and Sustainable*

project “VIRAL Data Engine – Virtualization and Intelligence Resource for Advanced Learning” (NRRP – M4C2, Investment 1.3, PE00000018; CUP: J33C22002910001), PRIN 2022 - ERC PE6 “TRex-SE: Trustworthy Recommenders for Software Engineers” (2022LKJWHC_03 - CUP: D53D23008730006) and PRIN 2022 - ERC SH5, PE6 “The Words of Peace and Pacifism. French Literature in the Interwar period by exploiting Distributional Semantic Analysis” (P20228AMFB - CUP: D53D23019570001)

REFERENCES

[1] E. Daka and G. Fraser, “A survey on unit testing practices and problems,” in *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*. IEEE Computer Society, 2014, pp. 201–211. [Online]. Available: https://doi.org/10.1109/ISSRE.2014.11

[2] G. Fraser and A. Arcuri, “Evosuite: automatic test suite generation for object-oriented software,” in *SIGSOFT/FSE’11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC’11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[3] V. Guilherme and A. Vincenzi, “An initial investigation of chatgpt unit test generation capability,” in *8th Brazilian Symposium on Systematic and Automated Software Testing, SAST 2023, Campo Grande, MS, Brazil, September 25-29, 2023*, A. L. Fontão, D. M. B. Paiva, H. Borges, M. I. Cagnin, P. G. Fernandes, V. Borges, S. M. Melo, V. H. S. Durelli, and E. D. Canedo, Eds. ACM, 2023, pp. 15–24. [Online]. Available: https://doi.org/10.1145/3624032.3624035

[4] 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. [Online]. Available: https://doi.org/10.1109/TSE.2023.3334955

[5] M. L. Siddiq, J. C. da Silva Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, “Using large language models to generate junit tests: An empirical study,” in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE 2024, Salerno, Italy, June 18-21, 2024*. ACM, 2024, pp. 313–322. [Online]. Available: https://doi.org/10.1145/3661167.3661216

[6] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, “No more manual tests? evaluating and improving chatgpt for unit test generation,” *CoRR*, vol. abs/2305.04207, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.04207

[7] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, “Chatgpt vs SBST: A comparative assessment of unit test suite generation,” *IEEE Trans. Software Eng.*, vol. 50, no. 6, pp. 1340–1359, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3382365

[8] G. Grano, F. Palomba, D. D. Nucci, A. D. Lucia, and H. C. Gall, “Scented since the beginning: On the diffuseness of test smells in automatically generated test code,” *J. Syst. Softw.*, vol. 156, pp. 312–327, 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2019.07.016

[9] M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy, “METHODS2TEST: A dataset of focal methods mapped to test cases,” in *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 299–303. [Online]. Available: https://doi.org/10.1145/3524842.3528009

[10] C. Csallner and Y. Smaragdakis, “Jcrasher: an automatic robustness tester for java,” *Softw. Pract. Exp.*, vol. 34, no. 11, pp. 1025–1050, 2004. [Online]. Available: https://doi.org/10.1002/spe.602

[11] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: https://doi.org/10.1109/ICSE.2007.37

[12] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, “GRT: program-analysis-guided random testing (T),” in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE Computer Society, 2015, pp. 212–223. [Online]. Available: https://doi.org/10.1109/ASE.2015.49

[13] A. Sakti, G. Pesant, and Y. Guéhéneuc, “Instance generator and problem representation to improve object oriented code coverage,” *IEEE Trans. Software Eng.*, vol. 41, no. 3, pp. 294–313, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2014.2363479

[14] J. H. Andrews, T. Menzies, and F. C. H. Li, “Genetic algorithms for randomized unit testing,” *IEEE Trans. Software Eng.*, vol. 37, no. 1, pp. 80–94, 2011. [Online]. Available: https://doi.org/10.1109/TSE.2010.46

[15] P. Derakhshanfar, X. Devroey, and A. Zaidman, “Basic block coverage for search-based unit testing and crash reproduction,” *Empir. Softw. Eng.*, vol. 27, no. 7, p. 192, 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10155-0

[16] P. Tonella, “Evolutionary testing of classes,” in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, G. S. Avrunin and G. Rothermel, Eds. ACM, 2004, pp. 119–128. [Online]. Available: https://doi.org/10.1145/1007512.1007528

[17] X. Devroey, A. Gambi, J. P. Galeotti, R. Just, F. M. Kifetew, A. Panichella, and S. Panichella, “JUGE: an infrastructure for benchmarking java unit test generators,” *Softw. Test. Verification Reliab.*, vol. 33, no. 3, 2023. [Online]. Available: https://doi.org/10.1002/stvr.1838

[18] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, “An empirical investigation on the readability of manual and generated test cases,” in *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, F. Khomh, C. K. Roy, and J. Siegmund, Eds. ACM, 2018, pp. 348–351. [Online]. Available: https://doi.org/10.1145/3196321.3196363

[19] S. Paydar and A. Azamnouri, “An experimental study on flakiness and fragility of randoop regression test suites,” in *Fundamentals of Software Engineering - 8th International Conference, FSEN 2019, Tehran, Iran, May 1-3, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, H. Hojjat and M. Massink, Eds., vol. 11761. Springer, 2019, pp. 111–126. [Online]. Available: https://doi.org/10.1007/978-3-030-31517-7_8

[20] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, “Learning deep semantics for test completion,” in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2111–2123. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00178

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

[22] H. A. Chipman, E. I. George, and R. E. McCulloch, “Bart: Bayesian additive regression trees,” *The Annals of Applied Statistics*, pp. 266–298, 2010.

[23] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, “Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 423–435. [Online]. Available: https://doi.org/10.1145/3597926.3598067

[24] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, “Fuzz4all: Universal fuzzing with large language models,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 126:1–126:13. [Online]. Available: https://doi.org/10.1145/3597503.3639121

[25] N. Nashid, M. Sintaha, and A. Mesbah, “Retrieval-based prompt selection for code-related few-shot learning,” in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2450–2462. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00205

[26] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan,

and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[27] F. Palomba, D. D. Nucci, A. Panichella, R. Oliveto, and A. D. Lucia, "On the diffusion of test smells in automatically generated test code: an empirical study," in *Proceedings of the 9th International Workshop on Search-Based Software Testing, SBST@ICSE 2016, Austin, Texas, USA, May 14-22, 2016*. ACM, 2016, pp. 5–14. [Online]. Available: https://doi.org/10.1145/2897010.2897016

[28] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, "A3test: Assertion-augmented automated test case generation," *Inf. Softw. Technol.*, vol. 176, p. 107565, 2024. [Online]. Available: https://doi.org/10.1016/j.infsof.2024.107565

[29] 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, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, M. d'Amorim, Ed. ACM, 2024, pp. 572–576. [Online]. Available: https://doi.org/10.1145/3663529.3663801

[30] 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, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 30–34. [Online]. Available: https://doi.org/10.1145/3639478.3640024

[31] Y. Wang, C. Xia, W. Zhao, J. Du, C. Miao, Z. Deng, P. S. Yu, and C. Xing, "Projecttest: A project-level LLM unit test generation benchmark and impact of error fixing mechanisms," *CoRR*, vol. abs/2502.06556, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2502.06556

[32] Q. Zhang, Y. Shang, C. Fang, S. Gu, J. Zhou, and Z. Chen, "Testbench: Evaluating class-level test case generation capability of large language models," *CoRR*, vol. abs/2409.17561, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2409.17561

[33] 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 *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, M. d'Amorim, Ed. ACM, 2024, pp. 185–196. [Online]. Available: https://doi.org/10.1145/3663529.3663839

[34] 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

[35] K. Jain, G. Synnaeve, and B. Rozière, "Testgeneval: A real world unit test generation and test completion benchmark," in *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. [Online]. Available: https://openreview.net/forum?id=7o6SG5gVev

[36] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A systematic survey of prompt engineering in large language models: Techniques and applications," *CoRR*, vol. abs/2402.07927, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.07927

[37] R. Garabík, "Processing xml text with python and elementtree–a practical experience," *Bratislava, L'. Štúr Institute of Linguistics*, 2005.

[38] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[39] M. Berglund and B. van der Merwe, "Formalizing BPE tokenization," in *Proceedings of the 13th International Workshop on Non-Classical Models of Automata and Applications, NCMA 2023, Famagusta, North Cyprus, 18th-19th September, 2023*, ser. EPTCS, B. Nagy and R. Freund, Eds., vol. 388, 2023, pp. 16–27. [Online]. Available: https://doi.org/10.4204/EPTCS.388.4

[40] M. Aniche, *Effective Software Testing: A developer's guide*. Simon and Schuster, 2022.

[41] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," *Adv. Comput.*, vol. 112, pp. 275–378, 2019. [Online]. Available: https://doi.org/10.1016/bs.adcom.2018.03.015

[42] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "tsdetect: an open source test smells detection tool," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 1650–1654. [Online]. Available: https://doi.org/10.1145/3368089.3417921

[43] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, 2001, pp. 92–95.

[44] G. Meszaros, S. Smith, and J. Andrea, "The test automation manifesto," in *Extreme Programming and Agile Methods - XP/Agile Universe 2003, Third XP and Second Agile Universe Conference, New Orleans, LA, USA, August 10-13, 2003, Proceedings*, ser. Lecture Notes in Computer Science, F. Maurer and D. Wells, Eds., vol. 2753. Springer, 2003, pp. 73–81. [Online]. Available: https://doi.org/10.1007/978-3-540-45122-8_9

[45] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: an exploratory study," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON 2019, Markham, Ontario, Canada, November 4-6, 2019*, T. Pakfetrat, G. Jourdan, K. Kontogiannis, and R. F. Enenkel, Eds. ACM, 2019, pp. 193–202. [Online]. Available: https://dl.acm.org/doi/10.5555/3370272.3370293

[46] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, C. S. Pasareanu and D. Marinov, Eds. ACM, 2014, pp. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[47] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[48] Y. Cui, "Webapp1k: A practical code-generation benchmark for web app development," *CoRR*, vol. abs/2408.00019, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.00019