Generating Project-Specific Test Cases with Requirement Validation Intention

BINHANG QI, National University of Singapore, Singapore
YUN LIN*, Shanghai Jiao Tong University, China
XINYI WENG, Shanghai Jiao Tong University, China
YUHUAN HUANG, Shanghai Jiao Tong University, China
CHENYAN LIU, National University of Singapore, Singapore
HAILONG SUN, Beihang University, China
ZHI JIN, Wuhan University, China and Peking University, China
JIN SONG DONG, National University of Singapore, Singapore

Test cases are valuable assets for maintaining software quality. State-of-the-art automated test generation techniques typically focus on maximizing program branch coverage or translating focal methods into test code. However, in contrast to branch coverage or code-to-test translation, practical tests are written out of the need to validate whether a requirement has been fulfilled. Specifically, each test usually reflects a developer's *validation intention* for a program function, regarding (1) *what is the test scenario of a program function?* and (2) *what is expected behavior under such a scenario?* Without taking such intention into account, generated tests are less likely to be adopted in practice.

In this work, we propose IntentionTest, which generates project-specific tests given the description of validation intention. The design is motivated by two insights: (1) **rationale insight**: the description of validation intention regarding scenario description and behavioral expectation, compared to coverage and focal code, carries more crucial information about *what to test*; and (2) **technical insight**: practical test code exhibits high duplication, indicating that existing tests are highly reusable for *how to test*. Therefore, IntentionTest adopts a retrieval-and-edit manner. First, given a focal code and a description of validation intention consisting of a test objective with test precondition and expected results, IntentionTest retrieves a reusable test in the project as the test reference. Then, IntentionTest edits the test reference with an LLM regarding the validation intention toward the target test. To guarantee that the target test can have a project-specific test prefix and a relevant test assertion, IntentionTest further explores the software project to identify *crucial code facts* (i.e., relevant API/code to call and global variables to refer to in the test) as important context for the test generation. We extensively evaluate IntentionTest against four baselines (TELPA, DA, ChatTester, and EvoSuite) on 3,680 test cases from 12 open-source projects. Compared to state-of-the-art baselines, with a given validation intention, IntentionTest can (1) generate tests far more semantically relevant to ground-truth

*Corresponding Author.

Authors' Contact Information: Binhang Qi, qibh@nus.edu.sg, National University of Singapore, Singapore, Singapore; Yun Lin, lin_yun@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Xinyi Weng, nanakusa@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Yuhuan Huang, hyh0u0@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Chenyan Liu, chenyan@u.nus.edu, National University of Singapore, Singapore, Singapore; Hailong Sun, sunhl@buaa.edu.cn, Beihang University, Beijing, China; Zhi Jin, zhijin@whu.edu.cn, Wuhan University, Wuhan, China and Peking University, Beijing, China; Jin Song Dong, dcsdjs@nus.edu.sg, National University of Singapore, Singapore, Singapore.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

tests by (i) killing 28.1% to 37.6% more common mutants and (ii) sharing 16.9% to 23.9% more common coverage; and (2) generate 23.7% to 49.0% more successful passing tests.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Test Generation, Software Testing, Large Language Model

ACM Reference Format:
Binhang Qi, Yun Lin, Xinyi Weng, Yuhuan Huang, Chenyan Liu, Hailong Sun, Zhi Jin, and Jin Song Dong. 2018. Generating Project-Specific Test Cases with Requirement Validation Intention. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software tests play a crucial role in ensuring the quality of both open-source and industrial software. They can be used for requirement clarification [31, 32], code review [7, 57], and CI/CD [12, 17, 22], ensuring the reliability of software products. The state-of-the-art software testing approaches can generally fall into the following categories:

- **Test Generation for Code Coverage.** Classical test generators [11, 20, 26, 28, 37] such as Klee [11], Dart [20], EvoSuite [18], Randoop [37], and their variants [26, 28] consider test generation as a problem of maximizing branch or path coverage. To this end, the problem of test generation is transformed into a problem of constraint solving, with solutions such as static and dynamic symbolic execution [9–11, 20, 46] and search-based software testing [5, 9, 18, 21, 26, 28, 29, 37].
- **Test Generation as Code Translation.** With the emergence of language models, LLM-based test generators [15, 25, 34, 50, 56] consider the test generation as a special case of code generation. Given a focal method and an instruction prompt, those techniques use LLMs (e.g., ChatGPT) to inductively generate tests by translating the prompt, focal method, or both to the test code.

While existing LLM-based test generators have shown promising results, practical test cases are not purely driven by coverage metrics or code-to-test translation. Instead, software developers write test cases by validating the consistency between the requirement and its implementation. Such a validation intention usually indicates (1) the test scenario of the program function and (2) the expected behavior of the program function. For example, a program function to start a server (see Listing 1) can be tested under a scenario where the option of using a thread pool. Its behaviors are validated by checking whether the pool is created as expected. Such intention can hardly be inferred from program branches (see the branchless code example in Listing 1). However, it is crucial to guide how we write the target tests.

In this paper, we use the term validation intention to denote the explicit articulation of what requirement to validate and how its fulfillment should be observed. Different from branch coverage, which focuses on structural exploration or code-to-test translation, which merely mirrors the focal method, validation intention captures the semantic link between requirements, usage scenarios, and expected behaviors. It thus serves as the conceptual bridge from requirement specification to executable tests. Concretely, validation intention can be instantiated as a semi-structured description (objective, preconditions, and expected results), but its essence lies in guiding test design towards meaningful requirement validation rather than superficial coverage.

According to our studies in a technical giant with over 10,000 software developers, regulated validation intention descriptions are adopted to write 67.8% of system tests and 18.6% of unit tests. However, converting validation intention into test code is technically challenging with the following knowledge gaps.

- **Implicit Project-Specific Test Idiom**: First, writing tests is more than just invoking the focal methods, which requires following specific project paradigms, including decisions such as "*How*

to construct the test inputs (e.g., constructor, factory, or singleton)?", "*What object to be mocked?*", and "*What resource to release after the test?*". As a result, additional project-specific code often appears before or after the invocation of the focal method in the test.

- **Implicit Project-Specific Constraints**: Second, test inputs can have very restrictive choices in the project. For example, in the Spark project [6], the range of parameter with name `headerType` of `int` type can only be limited to a few choices (e.g., 0, 1, 2, and 3, each stands for a type of message header) instead of falling into the range of -2^{n-1} to $2^{n-1}-1$ (n is the bit number).

In this work, we propose IntentionTest, which compiles the validation intention description into a *project-specific* test. The description of validation intention consists of a test objective, complemented with an optional test precondition and an expected result. Based on the empirical observation that test code is highly duplicated (see Section 3), we design IntentionTest to generate tests in a retrieval-and-edit manner. Specifically, given (1) a focal method m and (2) a validation intention description *desc*, IntentionTest generates tests in two stages, i.e., (1) in the **retrieval stage**, IntentionTest constructs m and *desc* as a query to search for an existing test as a reference and (2) in the **edit stage**, IntentionTest reduces the test generation problem into a code-editing problem on the reference. To edit the test reference into the correct target test, we explore the entire project for the crucial code facts (e.g., crucial project-specific APIs) based on (1) their semantics to the validation intention and (2) their historical relevance to the focal method (e.g., how often an API or a piece of code co-occurs with the invocation of the focal method). The crucial facts can significantly mitigate the LLM hallucination for reliable edits on the test reference. Finally, IntentionTest iteratively refines the generated test, resolving compilation errors, execution errors, and assertion failures until the test passes or the maximum iteration is reached.

We extensively evaluate IntentionTest against four baselines (i.e., TELPA [54], DA [47], ChatTester [56] and EvoSuite [18]) on 3,680 tests from 12 open-source projects. Compared to state-of-the-art approaches, i.e., TELPA, DA, and ChatTester, with a given validation intention, IntentionTest can (1) generate tests far more semantically relevant to the ground-truth tests by (i) killing 28.1% to 37.6% more common mutants and (ii) invoking up to 62.9% more project-specific APIs; (2) generate 36.7% to 49.0% more successful passing tests. Moreover, IntentionTest maintains its performance across both commercial LLMs (e.g., ChatGPT) and open-source LLMs (e.g., DeepSeek-V3.2). Finally, we evaluate the performance of IntentionTest across different granularities of the description of validation intention, demonstrating IntentionTest can still keep its performance even when we restrict the description by only keeping the test objective within 50 words.

In summary, this work makes the following contributions:

- **Methodology.** To the best of our knowledge, IntentionTest is the first solution to convert humans' validation intention (i.e., test requirement) to a project-specific test with both test prefix and oracle. IntentionTest is an agentic test generator supporting multiple granularities of intention descriptions, with stable test generation performance.
- **Empirical Study.** We conduct an empirical study over 12 open-source projects, showing that test reuse is prevalent in well-established projects. This finding provides a strong empirical foundation for the design of retrieval-and-edit-based test generation solutions.
- **Dataset & Tool.** We release the first benchmark where the focal code, test code, and validation intention (regarding the test objective, preconditions, and expected results) are available. We collect and curate 3,680 test cases across 12 projects to ensure high quality. Also, we deliver IntentionTest as a prototype tool in the form of a VS Code extension, visualizing the stepwise process (see tool video in [3]). The tool can facilitate practical test generation, particularly for well-established projects.

```
1public EmbeddedServer create(Routes rm, StaticFilesConfiguration conf, ExceptionMapper em
      , boolean han) {
2   MatcherFilter filter = new MatcherFilter(rm, conf, em, han);
3   filter.init(null);
4
5   JettyHandler handler = new JettyHandler(filter);
6   handler.getSessionCookieConfig().setHttpOnly(httpOnly);
7   return new EmbeddedJettyServer(serverFactory, handler).withThreadPool(threadPool);
8}
```

Listing 1. Focal method needs to be equipped with test cases in Spark project [6].

- **Evaluation.** We extensively evaluate IntentionTest against state-of-the-art baselines on our benchmark. Results show that IntentionTest can generate tests that are more semantically faithful to the validation intention while also outperforming baselines on general metrics, including mutation score, pass rate, and code coverage. Furthermore, we analyze the performance of validation intention across five levels of detail, demonstrating that it can still maintain the performance even when limited to a concise description of the test objective within 50 words.

Given the space limit, the demonstration video, tool source code, and more experimental results are available at [4].

2 Motivating Example

Listing 1 shows a focal method in the Spark project [6], which creates an embedded server, such as Jetty and Tomcat, in the Spark distributed system. Generating its test cases with the state-of-the-art solutions (e.g., EvoSuite [18], DA [47], and ChatTester [56]) for such focal code as in Listing 2 is challenging for the following reasons:

- **A Branch Cannot Represent A Test Scenario/Intention.** Despite that the focal method `create()` is short and branchless (see Listing 1), there are many validation options to test its different scenarios. For example, the server can be created with different options of thread pool and configured cookies. Therefore, full branch coverage can hardly indicate full scenario coverage. Without a specified validation intention (i.e., *create a server with thread pool*) as guidance, it is challenging to define and generate a *semantically* correct test.
- **Abundant Project Specifics.** As shown in Listing 2, the ground-truth test contains project-specific test prefixes and oracles. First, the test requires specific mocking since initializing or instantiating certain objects can be time- and resource-consuming (Lines 3-6). Second, a specific mocking condition (Line 8) must be defined before calling the focal method `create()`. Third, the test scenario requires calling two APIs (Lines 12-13) after invoking the focal method to ignite the server with a specific configuration, ensuring that the created server is runnable. Finally, unlike common assertions that simply check for the existence of created objects, this test scenario verifies the invocation time and the number of interactions by the created server (Lines 15-16).

Coverage-based Test Generators. For such a branchless focal method, traditional coverage-driven software testing lacks sufficient information to distinguish different test scenarios. In addition, although the generated tests are runnable, they often omit crucial project-specific setup and, by design, cannot reflect the developer's validation intention. Given the space limit, interested audiences can check an example of the EvoSuite-generated test at [4].

LLM-based Test Generators (Code-to-Test Translation). Listing 3 presents a test generated by ChatTester [56], a solution built on GPT-o1-mini. Even when fed with detailed validation intention from Table 1 as its additional prompt, ChatTester misses crucial project-specific pre- and post-conditions, including (1) specifying additional mocking behaviour (Line 8 in Listing 2) and (2) starting the server with specific configurations and generating assertions with specific intents (Lines 12-16 in Listing 2).

```
public void create_withThreadPool(){
    QueuedThreadPool pool = new QueuedThreadPool();
    JettyServerFactory jFactory = mock(...);
    StaticFilesConfiguration conf = mock(...);
    ExceptionMapper map = mock(...);
    Routes routes = mock(...);

    when(jFactory.create(pool)).thenReturn(new Server(pool));

    EmbeddedJettyFactory eFactory = new EmbeddedJettyFactory(jFactory).withThreadPool(
        pool);
    embeddedServer = eFactory.create(routes, conf, map, false);
    embeddedServer.trustForwardHeaders(true);
    embeddedServer.ignite("localhost", 6758, null, 0, 0, 0);

    verify(jFactory, times(1)).create(pool);
    verifyNoMoreInteractions(jFactory);
}
```

Listing 2. Target test corresponds to the focal method in Spark project. Codes in green are project-specific, which can hardly be inferred by LLMs learned from a general corpus.

```
public void create_withThreadPool() {
    Routes routeMatcher = mock(Routes.class);
    StaticFilesConfiguration staticFilesConfiguration = mock(StaticFilesConfiguration.
        class);
    ExceptionMapper exceptionMapper = mock(ExceptionMapper.class);
    ThreadPool pool = new QueuedThreadPool();
    EmbeddedJettyFactory factory = new EmbeddedJettyFactory().withThreadPool(pool);

    EmbeddedServer server = factory.create(routeMatcher, staticFilesConfiguration,
        exceptionMapper, false);

    assertNotNull("EmbeddedServer should not be null", server);
}
```

Listing 3. Test generated by ChatTester, missing project-specific pre- and post-setup.

Table 1. The validation intention description follows ISO/IEC/IEEE 29119 [1]. *Objective* section is mandatory, *Preconditions* (optional) and *Expected Results* sections are optional.

Objective:
Tests creating an embedded server with a custom thread pool and checks if the server starts correctly.
Expected Results:
1. Jetty server factory creates a server with the specified thread pool exactly once.
2. The server is initialized with the provided configurations and launched at port 6758 successfully.
3. No additional interactions with Jetty server factory occur beyond initial creation.

To overcome the above challenges, IntentionTest adopts a retrieval-and-edit solution to parse a focal method and a description of validation intention (see Table 1) to generate a test as shown in Listing 4. Specifically, IntentionTest (1) retrieves referable tests relevant to the target test scenario and (2) edits the referable test regarding the provided validation intention description. The code in red is where the edit location shall happen, and the code in green is the expected edit content in each location. Note that, the performance of IntentionTest is stable if we only keep the section of test objective and expected results in Table 1, which enables the programmers to spend minimal effort on writing the validation intention.

Challenges of Knowledge Gap. To generate a test as shown in Listing 4, IntentionTest needs to be aware of crucial code facts such as the definition of `JettyServerFactory.create(ThreadPool)`, `Server(ThreadPool)`, and `EmbeddedJettyFactory.withThreadPool(ThreadPool)` (see green text in Listing 4) from thousands of program elements in the project. Without those facts, LLM can easily hallucinate arbitrary non-existent APIs to invoke in the tests, causing compilation errors. However, the description of validation intention is in natural language form, sometimes

```
- public void create_withoutHttpOnly(){
+ public void create_withThreadPool(){
+    QueuedThreadPool pool = new QueuedThreadPool();
     JettyServerFactory jFactory = mock(...);
     StaticFilesConfiguration conf = mock(...);
     ExceptionMapper map = mock(...);
     Routes routes = mock(...);

-    Server server = new Server();
+    Server server = new Server(pool);

-    when(jFactory.create(100, 10, 10000)).thenReturn(server);
+    when(jFactory.create(pool)).thenReturn(server);

-    EmbeddedJettyFactory eFactory = new EmbeddedJettyFactory(jFactory).withHttpOnly(false);
+    EmbeddedJettyFactory eFactory = new EmbeddedJettyFactory(jFactory).withThreadPool(pool);

     embeddedServer = eFactory.create(routes, conf, map, false);
     embeddedServer.trustForwardHeaders(true);

-    embeddedServer.ignite("localhost", 6759, null, 100, 10, 10000);
+    embeddedServer.ignite("localhost", 6758, null, 0, 0, 0);

-    assertFalse(server.getHandler().getSessionCookieConfig().isHttpOnly());
+    verify(jFactory, times(1)).create(pool);
+    verifyNoMoreInteractions(jFactory);
```

Listing 4. Referable test retrieved and edited by IntentionTest towards the target test. The code in red is the test code to be modified, and the code in green is the expected test code.

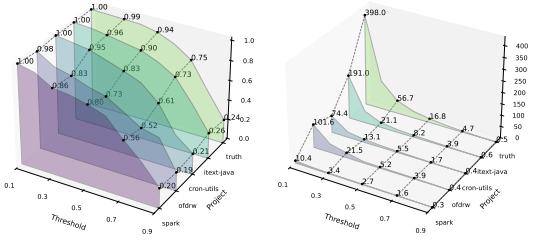
very different from the API names; thus, precisely mapping the description to concrete program elements is non-trivial. In addition, it is also non-trivial to adopt the identified facts into code edits on the test reference.

3 Empirical Study

In this section, we report our empirical study on open-source projects to evaluate our hypothesis that *existing test cases are highly reusable assets for writing a new test case*. The validated hypothesis lays an important foundation for our retrieval-and-edit design of IntentionTest.

3.1 Setup

Dataset. We collect 12 diverse popular open-source projects, spanning over web development, image processing, mobile development, and document generation conversion, each with over 100 GitHub stars and forks. For the initial dataset, *Dataset-FIX*, we snapshot a recent commit for each project to extract pairs of tests and their corresponding focal methods. This results in a total of 3,680 tests, with individual projects having between 51 and 1,099 tests. Each test has on average 25 lines of code (LoC), ranging from 9 to 124 LoC. Each focal method has on average 12 LoC, ranging from 3 to 331 LoC.



(a) Reference Availability (b) Referability Level

Fig. 1. The results of the empirical test referability. (a) and (b) present the RA and RL values and their trends across thresholds from 0.1 to 0.9 (as colored surfaces), with an interval of 0.1. Projects are sorted by the average number of candidates, from the fewest (155.8 for spark) to the most (2191.9 for truth). Even at high similarity threshold (e.g., 0.7), more than 50% tests have reusable references.

To support the empirical study's temporal requirements, we further derived *Dataset-TEM*. For each test t in *Dataset-FIX*, we reverted the project to the specific version where t was first created. We then identified all other tests existing at that point in time as potential retrieval candidates for t. Across the 12 projects, the average number of candidates per test ranges from 74.0 to 2191.9. Detailed statistics for both *Dataset-FIX* and *Dataset-TEM* are available in [4].

Metrics. We design two metrics: (1) **reference availability** for how likely a test can find its reference in its project? and (2) **reference level** for how many referable tests can a test have? Specifically, given a similarity threshold th and test similarity function $sim(.,.)$, we evaluate test referability as follows:

- **Reference availability**: Given the set of tests, T, we compute reference availability $RA_{th}(T)$ as:

$$RA_{th}(T) = \frac{\sum_{t \in T} \mathbf{1}(sim(t, T \setminus \{t\}) > th)}{|T|}, \tag{1}$$

  where $sim(t, T \setminus \{t\}) = \arg\max_{t^* \in T \setminus \{t\}} sim(t, t^*)$. Intuitively, $RA_{th}(T)$ measures the ratio of tests in T that have at least one non-identical test whose similarity exceeds th.
- **Referability level**: Given the set of tests, T, we calculate referability level $RL_{th}(T)$ as:

$$RL_{th}(T) = \frac{\sum_{t \in T} count(sim(t, T \setminus \{t\}) > th)}{|T|}. \tag{2}$$

$RL_{th}(T)$ evaluates the average number of referable tests per test. We use BM25 with normalization as the similarity metric, and vary the threshold from 0.1 to 0.9 (in increments of 0.1) to assess how different similarity cutoffs affect the results.

3.2 Results

Figure 1 shows the results of the empirical study on test referability. Overall, all 12 projects exhibit very high reference availability (RA) and referability level (RL). In Figure 1, we select 5 from the 12 projects based on the average number of candidates per test: `spark` [6], `ofdrw` [35], `cron-utils` [13], `itext-java` [24], and `truth` [49], with 155.8, 360.2, 475.8, 1279.4, and 2191.9 candidates, respectively. In Figure 1(a) and Figure 1(b), we show how the RA and RL value of each project under different thresholds, the higher the area under the curve, the more RA and RL value. For example in Figure 1(a), the `itext-java` project, 73% of the tests have references at the similarity threshold of 0.7. In Figure 1(b), even for the small project `spark`, $RL_{0.7}$ reaches 1.6, which means a target test has close to two references. Moreover, similar to the trend observed in RA, RL also exhibits an increase as the number of historical candidates grows. More details can be found in [4].

As a result, we conclude that (1) test referability in an open-source project is prevalent and (2) the more established a project, the more likely its tests can be referred to each other.

4 Approach

Problem Statement. Given a software project containing a set of tests T and a set of functions M, we assume each test $test \in T$ has a corresponding focal method $m \in M$ to test. The validation description $desc_{tar}$ is defined as a tuple $(obj, precondition, expectation)$ where (1) obj, $precondition$, and $expectation$ are textual description, (2) obj describes the test scenarios, $precondition$ describes what needs to be satisfied before running the focal method, and $expectation$ describes verifiable program behaviors, and (3) $precondition$ can be ϵ (i.e., empty string). Given a target focal method m_{tar} to be tested, and its validation description $desc_{tar}$, we generate the target test for m_{tar}, denoted as $test_{tar}$, regarding $desc_{tar}$. Note that $desc_{tar}$ is a semi-structured description consisting of a test objective, optionally complementary with test precondition and expected results.

Approach Overview. Figure 2 shows our approach consisting of the following stages:

- **Stage 1 (Referable Test Retrieval):** IntentionTest converts m_{tar} and $desc_{tar}$ into a query to retrieve a ranked list of method-test pairs, each consisting of a focal method m_{ref} and its test $test_{ref}$, from the project. $test_{ref}$ is also called as a referable test to edit into the target test $test_{tar}$.

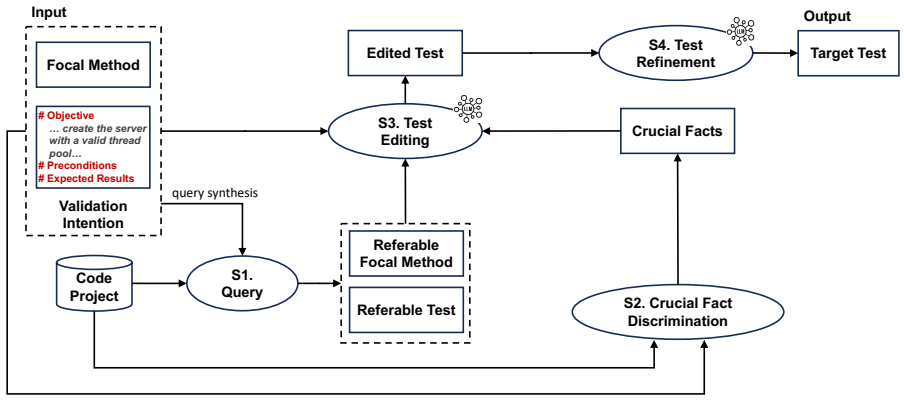


Fig. 2. Overview of IntentionTest: Given a focal method and a validation intention description of its test, IntentionTest derives the test in a retrieval-and-edit manner.

- **Stage 2 (Crucial Facts Discrimination):** Based on the validation intention $desc_{tar}$, IntentionTest explores the project, following a variety of program relationships (e.g., caller, callee, declaration, etc.), for identifying crucial facts, $facts$, to edit the referable test.
- **Stage 3 (LLM-based Test Editing):** IntentionTest constructs a prompt, incorporating general information such as $desc_{tar}$ and m_{tar}, as well as the project-specific information like m_{ref}, $test_{ref}$, and $facts$. Then, IntentionTest prompts the LLM to edit $test_{ref}$ towards $test_{tar}$.
- **Stage 4 (LLM-based Self Refinement):** Once $test_{tar}$ is derived, IntentionTest validates whether $test_{tar}$ compiles and executes successfully against m_{tar}. If errors occur, IntentionTest iteratively refines $test_{tar}$ to correct compilation and execution errors based on error messages and $facts$.

IntentionTest iterates these stages until $test_{tar}$ successfully passes, or a maximum number of iterations is reached.

4.1 Referable Test Retrieval

Given m_{tar} and $desc_{tar}$, IntentionTest retrieves from the project a referable focal method m_{ref} and its referable test $test_{ref}$ to edit. Intuitively, the fewer editing efforts to transform $test_{ref}$ into the target test $test_{tar}$, the higher the likelihood of accomplishing the editing. Technically, given a code similarity metric, denoted as $sim(.,.)$, we ideally have $test_{ref} = \arg\max_{test \in T} sim(test, test_{tar})$. However, $sim(test, test_{tar})$ is not computable when we retrieve $test_{ref}$, as $test_{tar}$ is unknown. Therefore, we define a referability estimation function (REF) on the focal method m_{tar}, validation intention $desc_{tar}$, candidate referable method m, and candidate referable test $test$ so that the similarity between $test$ and $test_{tar}$ can be estimated. In this work, given a pair of m and $test$, we heuristically estimate the test referability by (1) the similarity between m_{tar} and m; (2) the similarity between $desc_{tar}$ and the validation intention of $test$, denoted as $test.desc$:

$$REF(m_{tar}, desc_{tar}, m, test) = \alpha \cdot sim(m_{tar}, m) + (1 - \alpha) \cdot sim(desc_{tar}, test.desc). \quad (3)$$

In this equation, we let $\alpha \in (0, 1)$. As for the similarity between m_{tar} and $p.m$, we use a normalized BM25 score (with min-max normalization). Regarding the similarity between $desc_{tar}$ and $test.desc$, we calculate the cosine similarity between their embeddings that are obtained using a lightweight embedding model, i.e., a pretrained CodeT5+ Embedding model [51]. Note that, the summarized test objective, test precondition, and expected results allow us to compare tests in a more precise manner. Finally, the pair $(m, test)$ with the highest REF value is selected as the reference.

```java
public EmbeddedServer create(Object identifier,
                             Routes routeMatcher,
                             ExceptionMapper exceptionMapper,
                             StaticFilesConfiguration Config,
                             boolean multipleHandlers) {

    EmbeddedServerFactory factory = factories.get(identifier);

    if (factory != null) {
        return factory.create(routeMatcher, Config,
                              exceptionMapper,
                              multipleHandlers);
    } else {
        throw new RuntimeException(...);
    }
}
```

Rank 1

```java
private final JettyServerFactory serverFactory;
private Server server;
private ThreadPool threadPool = null;
.....
```

Rank 2

```java
public void extinguish () {
    logger.info(">>> {} shutting down ...", NAME);
    try {
        if (server != null) {
            server.stop();
        }
    } catch (Exception e) {
        logger.error("stop failed", e);
        System.exit(100); // NOSONAR
    }
}
```

Rank 3

Fig. 3. Taking the description of Table 1 as query (i.e., *Tests creating an embedded server...*), all the top-3 retrieved program elements by embedding similarity are irrelevant. See a relevant program element in Figure 5.

4.2 Crucial Facts Discrimination

Given the referable test $test_{ref}$ and the validation intention $desc_{tar}$, we then design an agentic project-wise editing approach to edit $test_{ref}$ to the target test. Specifically, the LLM agent needs to be aware of the program entities in the project for editing the test (see the code in green in Listing 4). For example, in Listing 4, IntentionTest shall identify (1) `QueuedThreadPool` is the only appropriate implementation of the `ThreadPool` interface (see Line 3 in Listing 4); (2) the overloaded method `Server(ThreadPool)` should be used to replace `Server()`; and (3) the overloaded method `JettyServerFactory.create(ThreadPool)` should be used to replace `JettyServerFactory.create(int, int, int)`. We call such program entities (along with their code) as *crucial fact* for the LLM agent to edit the referable test.

Challenge. However, a project can contain thousands of program entities while only a very small subset is relevant. Figure 3 shows a naive embedding-based approach to retrieve program elements regarding the intention description, As a result, none of the top-3 elements is relevant, which indicates that textual similarity is not sufficient for retrieving the most relevant program entities. False positive crucial facts can mislead our LLM agent to generate incorrect test code.

To address this, we first build a code graph where a node indicates a program entity (e.g., *class*, *method*, and *field*), and an edge indicates their relationship (e.g., *call* and *define*), for exploring crucial facts by discriminating program entities regarding both their *semantic* and *historical relevance* to the focal method m_{tar} and the validation intention $desc_{tar}$.

4.2.1 Code Graph Exploration. The goal of code graph exploration is to collect facts relevant to the focal method m_{tar}, the intention description $desc_{tar}$, and the referable test $test_{ref}$ for the target test $test_{tar}$ to adapt with. Given m_{tar}, $desc_{tar}$, and $test_{ref}$, IntentionTest starts its exploration from both m_{tar} and $test_{ref}$, than traversing the project's code graph. In the graph, each node represents a program entity, including *class*, *interface*, *method*, and *field*; each edge represents a relationship between entities, including *define*, *call*, *param*, *overload*, *implement*, and *extend*. Note that, for each directed edge (e.g., call), we also derive a new edge as its reverse-direction relation (e.g., called by). The maximum exploration depth from m_{tar} and $test_{ref}$ is bounded by a user-defined threshold.

4.2.2 Fact Discrimination. For each visited program entity, we evaluate its relevance by calculating its semantic and historical relevance to the focal method m_{tar} and the validation intention $desc_{tar}$. While semantic relevance is still calculated based on embedding similarity, we calculate historical relevance based on how likely a program entity can be used together with the focal method. Intuitively, we consider a test case of a focal method m_{tar} as a special case of code usage examples. Therefore, to find the crucial facts for writing the test of m_{tar}, we check all the existing usage

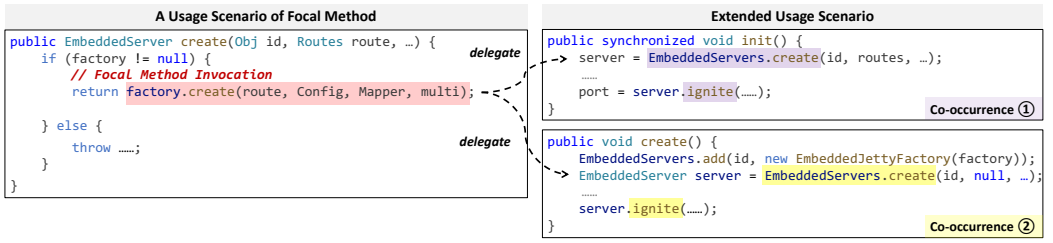


Fig. 4. An example of extending usage scenarios. By discovering the method delegation relation, we can extend more usage scenarios to find the co-occurring program entities (a.k.a., `ignite()`) with the focal method (a.k.a, `create(...)`).

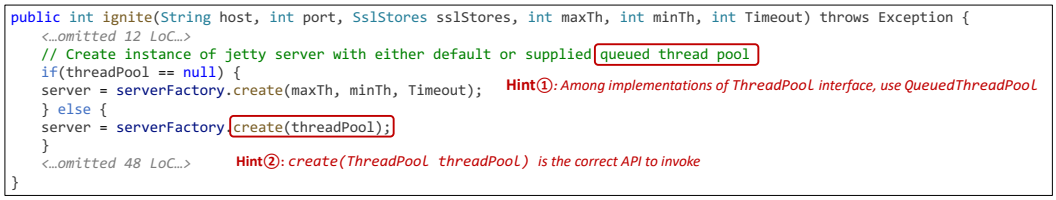


Fig. 5. Crucial fact `ignite()` provides two hints for editing the test case shown in Listing 4.

examples in the project for the frequent co-occurring entities of m_{tar}. Technically, given a set of candidate facts F (by traversing the code graph) and focal method usages U, we rank F by the likelihood of each candidate $f \in F$ by aggregating their quantified semantic and historical relevance. Then, the top k candidates are selected as the crucial facts set $F_{crucial}$.

Semantic Relevance Measurement. Given the intention description $desc_{tar}$ (e.g., *"...creating an embedded server with a custom thread pool..."* in Table 1) and a program entity c (c includes both code signature and body), we use a pre-trained code embedding model (e.g., CodeT5+ [51]) to have their embeddings, denoted as $e(desc_{tar})$ and $e(c)$. Then we take their cosine similarity, i.e., $cos(e(desc_{tar}), e(c))$, as their semantic relevance.

Historical Relevance Measurement. To mitigate the false positives and false negatives by embedding similarity, we measure the historical relevance of a program entity to the focal method, regarding their co-occurring likelihood. To this end, we take two steps, i.e., co-occurrence identification and weighted co-occurrence calculation.

Co-occurrence Identification. We first identify all the usage scenarios of the focal methods m_{tar}, where each scenario is a method invoking m_{tar}. Given a usage scenario s_i of m_{tar}, we call the set of all its invoking program elements as a **co-occurring set** to m_{tar}, denoted as $co\_occur(s_i, m_{tar}) = \{e_1, e_2, ..., e_k\}$ where e_i is a **co-occurring element** with m_{tar}.

In addition, we further expand the usage scenarios regarding the method delegation relation as shown in Figure 4, where the left rectangle represents a usage scenario of the focal method `create()` (in red background). In this example, the usage scenario of `create(Obj, Routes, ...)` can be extended to the usage scenario of `init()` and `create()` (see the right rectangles in Figure 4) because the focal method `create()` is a *delegation* of its caller method `create(Obj, Routes, ...)`. In other words, the method `create()` and the method `create(Obj, Routes, ...)` have strong functional coherence despite their syntactic difference. Here, we define a method m_i is the **delegation** of a method m_j if (1) m_j calls m_i and (2) m_j takes the returned variable of m_i as the

Table 2. The prompt template for test editing. The content in red text is to be filled in interactively.

#Target Focal Method: [code of target focal method]
#Target Focal Method Context: [skeleton of target focal file]
#Target Test Case: // A JUnit [version] test case to be generated
#Target Validation Intention Desc: [Objective, Preconditions, and Expected Results]
#Referable Test Case: [code of referable test]
#Crucial Project Knowledge: [list of crucial facts]
#Instruction: Please generate ONE #Target Test Case# for #Target Focal Method# by strictly following #Target Validation Intention Desc# and referring to #Referable Test Case# and #Relevant Project Information#....
#Requirements: Your final output must contain only ONE test method annotated '@Test' and strictly adhere to the following format: (1) Begin with the exact prefix: "```package ". (2) End with the exact suffix: "```".

returned variable. Then, for each delegated method m_d of the focal method m_{tar}, we can take the co-occurring set of m_d as the expanded co-occurring set of m_{tar}.

Example. For the focal method `create(...)` in Figure 4, we can have its original co-occurring set $cset_1 = \emptyset$ as there is no co-occurring program entities to `create(...)`. Nevertheless, it has two expanded co-occurring sets, from `init()` and `create()` respectively, i.e., $cset_2 = \{$`ignite(...)`$\}$ and $cset_2 = \{$`add, ignite(...)`$\}$.

Weighted Co-occurrence Calculation. For each co-occurring element e_i, we calculate its co-occurring probability to m_{tar} by:

$$occu_i = \sum_{cset_j \in CSet} \frac{sim(cset_j, desc_{tar}) \cdot \mathbf{1}(e_i \in cset_j)}{|CSet|}, \tag{4}$$

where (1) $\mathbf{1}(e_i \in cset_j)$ returns 1 if the co-occurring element e_i is in a co-occurring set $cset_j$, and 0 otherwise, (2) U is the set of co-occurring sets, and (3) the embedding similarity (i.e., by cosine similarity) between a co-occurring set and the validation intention is used to weight each co-occurrence. Finally, the likelihood of f_i being a crucial fact is calculated as:

$$\mathcal{L}_i = \beta \times sim_i + (1 - \beta) \times occu_i, \tag{5}$$

where $0 < \beta < 1$. Finally, IntentionTest selects program entities with their likelihood above a user-defined threshold as the crucial fact.

4.3 Test Editing and Refinement

Table 2 presents the prompt template for the LLM to edit the referable test, which includes m_{tar}, $desc_{tar}$, $test_{ref}$, and $F_{crucial}$. In the prompt, we parameterize some instructions such as *Target Focal Method* and *Referable Test Case*. In addition, we enforce the LLM to focus on collected crucial facts (i.e., *Crucial Project Knowledge*).

Given an edited test, IntentionTest compiles and executes the test, collecting feedback messages from the compiler and interpreter. IntentionTest extracts error messages and filters out irrelevant messages that involve files outside the project (e.g., code files from dependency packages). Then, IntentionTest constructs a prompt to refine the edited test, addressing the errors. The prompt for refinement is based on the prompt for test editing with modifications: (1) adding the edited test and extracted error messages, and (2) changing "#Instruction" part to request revision of the edited test. IntentionTest iteratively refines the edited test until it passes successfully or a maximum number of iterations is reached (e.g., 4 rounds). More details about the prompts are available at [4].

5 Evaluation

We evaluate IntentionTest with three research questions:
RQ1 (Overall Performance): How effective is IntentionTest in generating project-specific tests compared to the state-of-the-art test generators?
RQ2 (Granularity of Validation Intention): How does the granularity of intention description affect IntentionTest's performance? Or, how many details/efforts do the programmers need to provide to generate acceptable tests?
RQ3 (Ablation Study): How do referable tests (i.e., retrieval) and crucial facts (i.e., crucial fact discrimination) contribute to overall performance?

5.1 Overall Performance (RQ1)

5.1.1 Dataset. We use the 3,680 test cases, as the ground-truth, from 12 open-source repositories (see Section 3) to evaluate the performance of IntentionTest.
Validation Intention Generation. Due to the scarcity of explicit requirement documentation in open-source repositories, we synthesized validation intentions to facilitate a robust evaluation. To ensure experimental diversity and rigor, we generated these descriptions via two strategies:

- **LLM-Inferred**: For each focal method, we prompt GPT-o1-mini to propose candidate validation intentions without access to the corresponding test. We then align each intention to an existing test using an LLM-based matching step; we discard cases where no clear correspondence is found. This yields 2,536 test–intention pairs.
- **Human-Written**: We recruited six experienced developers to author validation intentions for 40 focal methods of 9 projects based only on the code implementation. The participants then mapped these intentions to relevant tests, resulting in a curated subset of 86 tests with human-validated intention descriptions.

Commit Timeline Reconstruction. To prevent temporal leakage in retrieval, we reconstruct each test's creation time and restrict retrieval candidates to tests whose last modification predates the target test's creation.
Mitigation of LLM's Memorization Effect. To avoid the inflated performance caused by the fact that LLMs could sometimes already *memorize* the target test code, we adopt an in-context unlearning technique [48] for LLMs to "forget" the test code details. Specifically, once we find that a naive prompt (e.g., just given the project name) can generate textually similar test code to that of a target test, we follow Pawelczyk et al's in-context unlearning approach [40] by introducing an unlearning prompt forcing LLM to ignore the whole knowledge of the project.

5.1.2 Metrics. We evaluate IntentionTest against the baselines upon the following metrics for (1) the syntactic quality of tests and (2) the semantic relevance of tests to the validation intention.

- **Compilation Failure, Execution Failure, Assertion Failure, and Successful Pass**: We track the rates of compilation failures, execution failures, assertion failures, and successful passes of the generated tests. An execution failure refers to test crashes at runtime, e.g., loading a nonexistent resource. An assertion failure refers to test's failing its assertion.
- **Common Mutation Score**: To quantify how much a generated test satisfies the given validation intention, we introduce the *common mutation score* (CMS) to compare the generated test and the ground-truth test to recover. Given a generated test t which can pass the assertion, we compare t with the corresponding ground-truth test t^*, let S_t and S_{t^*} denote the sets of mutants each test kills. CMS is calculated via the Jaccard Index over these sets: $CMS(t) = JI(S_t, S_{t^*}) = \frac{|S_t \cap S_{t^*}|}{|S_t \cup S_{t^*}|}$, where $CMS(t)$ ranges from 0 (totally different mutant-killing behaviour) to 1 (identical mutant-killing behaviour). Higher CMS values indicate stronger *intention alignment* between t and t^*. The

Table 3. Performance of IntentionTest and baselines (in %). "Evo." and "Chat." represent EvoSuite and ChatTester, respectively.

Metric		LLM-Inferred (2536 tests)						Human-Written (86 tests)					
		Evo.	Chat.	DA	TELPA	Ours(DS)	Ours(O1)	Evo.	Chat.	DA	TELPA	Ours(DS)	Ours(O1)
Compilation Failure		0.08	26.17	35.84	29.44	4.38	3.55	0.00	41.43	24.32	29.33	2.60	3.90
Execution Failure		0.00	4.42	4.06	4.32	2.80	2.96	0.00	2.86	12.16	12.00	5.19	1.30
Assertion Failure		7.87	1.08	17.10	2.12	3.11	1.50	15.07	4.29	18.92	1.33	3.90	0.00
Successful Pass		92.04	68.34	43.00	64.12	89.72	92.00	84.93	51.43	44.59	57.33	88.31	94.81
Exact Match (on coverage)		42.42	58.85	52.31	67.15	87.24	87.56	23.29	27.78	63.64	72.09	64.62	70.42
Common Coverage Ratio		66.52	70.27	69.48	76.50	92.91	93.40	58.16	47.93	81.49	85.40	83.72	89.28
CMS		30.32	38.72	41.82	48.26	55.30	76.35	35.62	36.25	42.39	50.10	60.53	85.18
CMS_{pair}	Baseline	29.57	51.94	52.61	69.41	-	-	36.75	34.88	59.93	74.26	-	-
	Ours	73.35	74.53	73.69	73.63	-	-	85.41	86.09	85.87	81.11	-	-
CodeBLEU		27.16	45.93	72.70	55.04	46.87	41.73	29.77	44.27	78.80	58.65	50.41	44.42

performance of a test generator in CMS is measured by the average CMS over all passing tests: $CMS(T) = \frac{1}{|T|}\sum_{t\in T} CMS(t)$. Note that, different approaches can have different performance in generating the assertion-passing tests. Therefore, we further introduce CMS_{pair} to compare IntentionTest with a baseline on their shared assertion-passing tests.

- **Line Coverage Overlap**: For each generated test that executes successfully, we compute line coverage on the focal method and compare it with the ground-truth test: (1)*Exact match* indicates identical covered-line sets; (2) *Common coverage ratio* is the Jaccard similarity between covered-line sets. Similarly to CMS, the proportion of generated tests that achieve these is used to measure alignment with validation intention.
- **CodeBLEU**: We also use CodeBLEU [43] to evaluate the similarity between the generated and ground-truth test cases.

5.1.3 Baselines. We choose ❶ ChatTester [56] and ❷ TELPA [54] as state-of-the-art agentic baselines for its outstanding performance among LLM-based test generators. To ensure a fair comparison, we upgraded the LLM used by them to o1-mini. TELPA is designed to generate tests for all callable focal methods in a project, and a focal method can have multiple generated tests. For a fair comparison, we select the best-performing generated test that has the highest CMS value among the generated tests for the focal method. We choose ❸ DA [47] as a state-of-the-art training-based baseline. Following DA's settings, we fine-tune its CodeT5 model on our dataset and evaluate its performance on our benchmark. We split the tests in each repository into two sets, each used as the training and the testing set for once. This ensures that all tests can be evaluated. Moreover, we choose ❹ EvoSuite [18] as the state-of-the-art search-based baseline, since its authors are actively incorporating new search strategies in the tool. As EvoSuite generates multiple tests for a focal method, we select a single test whose code coverage is most similar (measured by F1 score) to the ground-truth test. We do not use CMS for selection due to the trivial assertions of EvoSuite's generated tests.

We do not include IntUT [33] as a baseline, despite it having a seemingly similar idea to IntentionTest. IntUT uses generated validation intentions to guide LLMs in test generation, where *each intention targets a specific branch* in the focal method by specifying input parameters and expected outputs. As a result, IntUT is primarily designed to maximize code coverage. Moreover, its expected outputs are derived directly from the focal method, which assumes that the focal method returns constant values. Finally, neither the source code nor the dataset used by IntUT is publicly available, making a fair and reproducible comparison infeasible.

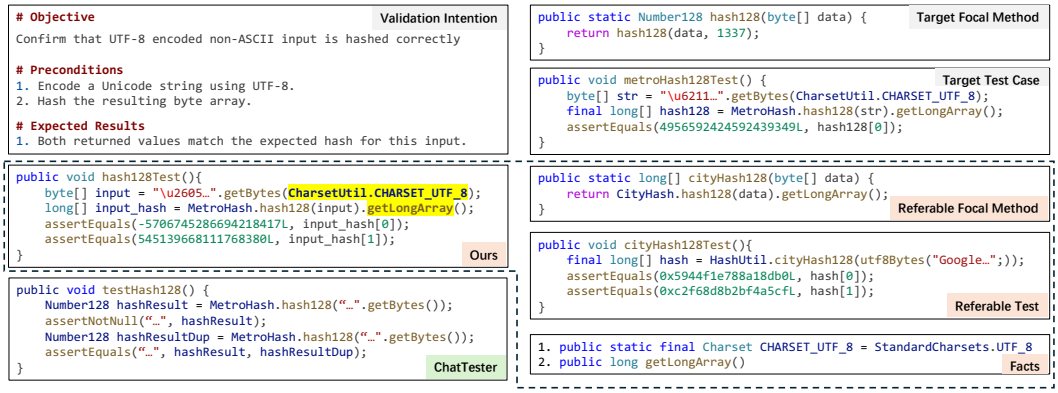


Fig. 6. Example comparison of generated tests for the same validation intention: IntentionTest vs. ChatTester. IntentionTest achieves higher alignment with the ground truth (CMS: 100% vs 8%).

5.1.4 Results. Table 3 summarizes all metrics with average performance across projects in the two datasets (i.e., a dataset using LLM-Inferred validation intention and another one using Human-Written validation intention). The column "Ours (DS)" reports results run upon DeepSeek-V3.2, and "Ours (O1)" upon o1-mini. In the experiment, IntentionTest can find a referable test in 69.01% and 75.32% of cases of test generation on "LLM-Inferred" and "Human-Written" datasets, respectively. A detailed breakdown of performance comparison across key metrics is available at [4]. Overall, IntentionTest outperforms the LLM-based baselines and performs on par with EvoSuite in general metrics (i.e., compilation failure, execution failure, assertion failure, and successful pass).

Performance on General Metrics We observe that DA, trained upon a small language model (i.e., CodeT5), struggles with syntax errors [47]. Moreover, EvoSuite leads the performance of generating passing test cases on most projects as it adopts regression oracles. IntentionTest significantly outperforms TELPA, ChatTester, and DA in success pass rate by 27.88%, 23.66%, and 49.00% on "LLM-Inferred" dataset. The results on "Human-Written" dataset consistently demonstrate the superiority of IntentionTest, with 37.48%, 43.48%, and 50.22% over TELPA, ChatTester, and DA, respectively. Upon investigation, we find that compilation and execution errors by the baselines (i.e., TELPA, DA, and ChatTester) are often caused by missing project-specific details. For example, many compilation errors stem from calling non-existent methods or missing project-specific packages.

Performance on Semantic Relevance As shown in Table 3, IntentionTest outperforms the baselines regarding line coverage overlap (including exact match and common coverage ratio) and common mutation score. Specifically, IntentionTest achieves the highest line coverage overlap (87.56% and 93.40%) and CMS results (76.35%), outperforming all state-of-the-art baselines on "LLM-Inferred" dataset, e.g., TELPA by 20.41%, 16.90%, and 28.09%, respectively. The results of "CMS_{pair}" further show the comparison of IntentionTest against each baseline on their common passing tests. IntentionTest consistently outperforms all baselines in validation intention alignment—for example, IntentionTest outperforms TELPA by 6.85% (81.11% vs. 74.26%) and 4.22% (73.63% vs. 69.41%) on "Human-Written" and "LLM-Inferred" dataset, respectively.

As for CodeBLEU, the results of IntentionTest, ChatTester, and TELPA are much smaller than DA's result by about 30%. Our results show that DA achieves a significant fit to the distribution of the 12 projects, and also imply the effectiveness of the strategy for mitigating the LLM's memorization effect. Moreover, we find that many generated tests by IntentionTest are semantically correct but syntactically different from the ground-truth.

Table 4. Performance of IntentionTest across five levels of intention description detail (in %).

Metric		LLM-Inferred					Human-Written				
		full	**obj**	**obj&pre**	**obj&exp**	**none**	**full**	**obj**	**obj&pre**	**obj&exp**	**none**
Compilation Failure		3.55	3.62	3.03	3.39	2.84	3.90	1.32	1.30	3.90	0.00
Execution Failure		2.96	2.88	2.80	2.88	2.99	1.30	2.63	0.00	2.60	0.00
Assertion Failure		1.50	1.81	1.65	2.46	1.38	0.00	1.32	3.90	3.90	1.30
Successful Pass		92.00	91.69	92.52	91.27	92.79	94.81	94.74	94.81	89.61	98.70
Exact Match (on coverage)		87.56	87.41	87.75	87.11	71.53	70.42	75.71	68.57	72.06	57.53
Common Coverage Ratio		93.40	93.38	93.55	93.31	84.54	89.28	88.91	87.18	87.83	75.59
CMS		76.35	70.49	72.03	71.47	68.26	85.18	74.15	73.79	73.80	71.52
CMS_{pair}	Variant	-	73.02	74.07	71.96	57.79	-	76.11	73.63	74.35	58.77
	Full	-	72.52	73.29	72.56	72.93	-	85.52	85.77	85.56	86.49
CodeBLEU		41.73	41.83	41.62	41.63	41.53	44.42	47.60	45.84	44.45	41.96

Figure 6 presents a case where IntentionTest significantly outperforms all baselines. The intended test scenario is to validate that the focal method `hash128()` correctly hashes a UTF-8-encoded string. The tricky parts are twofold: (1) *what to test*: recognize that the input should be encoded in UTF-8 and know the correct output; and (2) *how to test*: two crucial facts, i.e., `CHARSET_UTF_8` and `getLongArray()`, are required for the pre- and post-setup. Without the intention description and the facts, the generated tests fail their assertions (e.g., DA) or do not align with the validation intention (e.g., ChatTester and EvoSuite). For example, ChatTester uses the platform-default charset via `getBytes()` (which may not be UTF-8) and employs trivial assertions that do not verify the value returned by `getLongArray()` against the correct result, yielding a CMS of only 8%. In contrast, IntentionTest retrieves a reference and collects the required facts—both semantically close to the intention and frequently co-occurring with the focal method—achieving a CMS of 100%. More details are presented at [4].

Performance on Different LLMs. The column "Ours (DS)" in Table 3 shows that IntentionTest powered by DeepSeek-V3.2 still outperforms the baselines. Generally, the retrieved test reference and crucial code facts provide useful and enriched external knowledge, making our approach less dependent on the capability of foundation models. We also observe that IntentionTest (DS) underperforms IntentionTest (O1), caused by its limited inference performance of adopting crucial code facts in editing the referable test.

As a result, by capturing project-specific information and integrating the description of validation intention, IntentionTest establishes itself as a new state-of-the-art approach for practical projects.

5.2 Granularity of Validation Intention (RQ2)

5.2.1 Setup. Given the effort required to write a validation intention description, real-world projects may impose different levels of detail for it. We investigate the effectiveness of the description across five levels of detail, including full description (*full*), objective only (*obj*), objective & preconditions (*obj&pre*), objective & expected results (*obj&exp*), and no description (*none*).

5.2.2 Results. As shown in Table 4, providing any component can improve performance (excluding metrics related to successful pass), indicating the effectiveness of the three components. Notably, even providing only a test objective within 50 words can significantly improve LLMs in validation intention alignment, yielding CMS_{pair} improvements of 15.1% and 27.7% over IntentionTest without intention description (72.9% vs 57.7% and 86.5% vs 58.8%) and 4.2% and 6.9% over TELPA (73.6% vs 69.4% and 81.1% vs 74.2%).

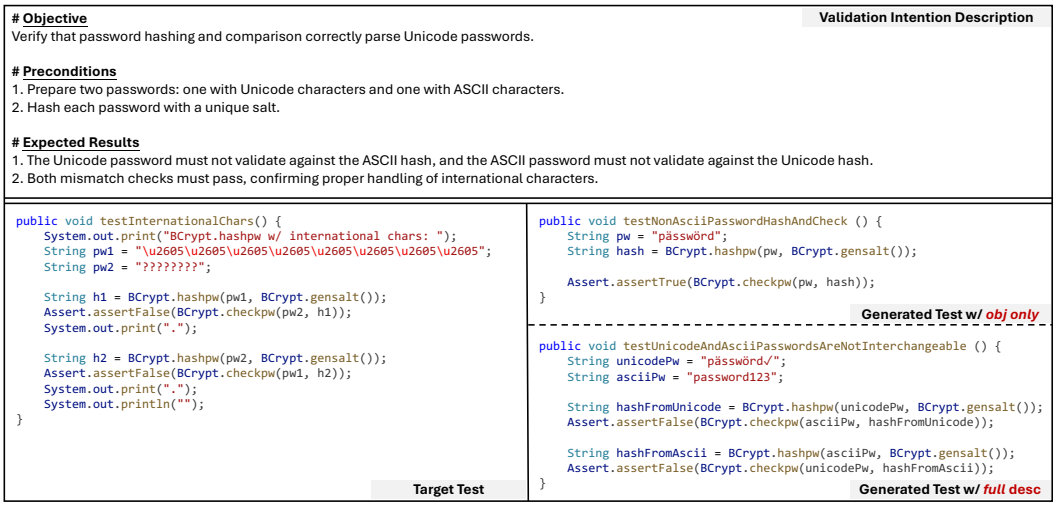


Fig. 7. Illustrative example of validation intention granularity: target test `testInternationalChars` and IntentionTest outputs generated using the full intention description and the objective-only description.

This improvement is expected, as even a brief Objective description captures the developer's core intention and helps guide test generation accordingly. For example, Figure 7 presents a target test `testInternationalChars` from the blade project [8], alongside its validation intention description (from "LLM-Inferred" dataset) and the corresponding tests generated with the full description and the Objective only, respectively. The target test primarily verifies correct hashing of non-ASCII passwords. It also incorporates a personalized intention: validating hashing and comparison correctness for a password consisting of black stars (i.e.,`u2605`) by cross-checking results with another password, rather than simply comparing plain and hashed values. In this case, the Objective portion of the generated description—"Verify that password hashing and comparison correctly parse Unicode passwords"—succinctly captures the main intention. Consequently, the test generated with the Objective only already aligns with the main intention, using a password containing non-ASCII characters "ä" and "ö". We also observed that the test generated with the full description aligns even more closely, as the Preconditions and Expected Results provide complementary details reflecting the personalized intention. In contrast, without such an intention description, ChatTester, and DA instead generate tests with common ASCII passwords, failing to capture the intended validation goal. These results highlight the practicality of validation intention descriptions, as even a concise Objective can substantially improve validation intention alignment.

We also analyze suboptimal generated tests to understand their causes. One common issue arises when the validation intention description is overly general. For example, a target test `codePointsAllIncludedRange` from project yavi [55] defines a whitelist by a project-specific API `CodePoints.Range.of()`, restricting input characters to a–z and A–Z. Due to our strict constraints on reverse engineering intention descriptions (e.g., minimizing inclusion of program elements), the derived description simplifies this intention to: "Define a whitelist which allows only uppercase and lowercase letters." As a result, IntentionTest generates a test that implements this intention naively by enumerating uppercase and lowercase letters in a list. While the generated test executes successfully, it partially misaligns with the target test. This misalignment is reflected in the CMS metric, as some mutants associated with `CodePoints.Range.of()` remain surviving. This result exposes a limitation of IntentionTest—the suboptimal ability of fact discrimination component to capture fine-grained semantic distinctions between codes and intention descriptions. We attribute

Table 5. Ablation study results for IntentionTest (in %).

Metric		LLM-Inferred			Human-Written		
		Full	No Fact	No Ref	Full	No Fact	No Ref
Compilation Failure		3.55	4.41	5.76	3.90	2.60	7.79
Execution Failure		2.96	3.55	5.64	1.30	1.30	2.60
Assertion Failure		1.50	1.30	1.18	0.00	5.19	3.90
Successful Pass		92.00	90.74	87.41	94.81	90.91	85.71
Exact Match (on coverage)		87.56	87.65	86.53	70.42	69.57	68.75
Common Coverage Ratio		93.40	93.61	92.84	89.28	87.43	84.08
CMS		76.35	72.44	73.90	85.18	78.80	74.18
CMS_{pair}	Ablation	-	72.65	68.31	-	73.23	65.85
	Full	-	72.94	72.72	-	84.72	84.47
CodeBLEU		41.73	41.19	34.73	44.42	45.12	35.14

this limitation partly to the weaker semantic understanding of the lightweight embedding model used (i.e., CodeT5 Embedding). Addressing this challenge is left for future work.

Furthermore, even without a validation intention description, IntentionTest still outperforms the state-of-the-art ChatTester—showing, for example, 29.54% and 35.27% gains in CMS (see the columns "ChatTester" in Table 3 and the columns "none" in Table 4). This result underscores the standalone effectiveness of our referable-test retrieval and crucial-facts discrimination methods. We also observe a small increase in successful passes when descriptions are removed. This occurs because the generation now faces fewer constraints, making it easier for to generate. However, this ease comes at the cost of lower CMS, indicating weaker alignment with the intended validation.

More experiments for IntentionTest powered by DeepSeek-V3.2, available at [4], also support the above conclusion.

5.3 Ablation Study (RQ3)

5.3.1 Setup. To assess the contribution of the retrieval module, we remove referable tests from the prompts. Regarding the discriminator's contribution to overall performance, we ablate it by removing facts from the prompt.

5.3.2 Results. As reported in Table 5, the column "*No Ref*" presents the performance results when the retrieval function is ablated. Compared to IntentionTest, the test generation performance declines sharply. For example, the performance in CMS drops by 2.45% and 11.00% on "LLM-Inferred" and "Human-Written" datasets, respectively. Similar degradations occur in all other metrics. The results indicate that referable tests can help reveal implicit project-specific test idioms and test scenarios that the LLM alone cannot infer.

We also evaluated the robustness of IntentionTest by analyzing its performance across a spectrum of similarity bins (i.e., $[0.2, 0.4]$, $(0.4, 0.6]$, $(0.6, 0.8]$, and $(0.8, 1.0]$) calculated using the Levenshtein-based edit similarity between the target and referable tests. We observed that, while higher similarity correlates with improved generation quality, IntentionTest maintains robust performance even in low-similarity regimes, consistently outperforming the *"No Ref"* variant. For example, in the Successful Pass metric, IntentionTest achieved 92.68%, 95.83%, 97.06%, and 96.15%, respectively. This suggests that (1) even low-similarity references could provide useful structural scaffolds and project-specific idioms and (2) LLM is relatively robust to irrelevant information.

As for the contribution of crucial facts, the column "*No Fact*" shows that removing facts from prompts likewise impairs performance, e.g., 3.91% and 6.38% of degradation in CMS on the two datasets. This confirms the effectiveness of the discriminator. Experiments for IntentionTest powered by DeepSeek-V3.2, available at [4], also support the above conclusion.

6 Discussion

6.1 Limitations and Future Work

6.1.1 Quality of Validation Intention Descriptions. In this work, validation intention descriptions generated by LLMs or written by developers are matched with ground-truth tests, filtering out low-quality and misaligned intentions. We attempted to make them as natural as possible—applying various constraints on generation and performing manual checks—yet a concern remains: human-written descriptions can be incomplete and ambiguous. Determining when a description is low quality and guiding users to craft higher-quality descriptions remains open work for future research.

6.1.2 Generalizability on Newly Established Repositories. In this paper, we target well-established repositories with sufficient human-written tests. The performance of IntentionTest might not generalize to newly established repositories. Although we evaluated IntentionTest on repositories with relatively few tests (e.g., 49 for *spark*), its effectiveness on repositories whose test suites are both sparse and low-quality is still unknown. Quantifying IntentionTest's usefulness in early-stage projects constitutes important future work.

6.2 Threats to Validity

The first threat involves the synthesized validation intention descriptions. Due to the limited availability of human-written validation intentions in open-source projects, we use LLMs to generate validation intention descriptions for existing tests. However, these synthesized descriptions may differ from those that developers would write. To mitigate this, we curate a set of human-written validation intention descriptions as a complementary dataset.

The second threat involves potential data leakage from LLMs, a common issue also faced by related work [56]. The tests in our evaluation dataset could be part of training data in LLMs, which might lead to overestimation of LLM's capability in test generation. To mitigate this threat, we design a prompt that instructs LLMs to "unlearn" the memorized information from the specified repository. Experimental observations confirm its effectiveness in preventing the generation of program elements (e.g., method invocations) that do not appear in the input context.

Finally, a threat arises from the nondeterministic nature of LLMs. To alleviate this, we set the temperature parameter to zero, constraining randomness and ensuring the most deterministic response possible [36].

7 Related Work

Coverage-based software testing. Software testing is traditionally considered a constraint-solving problem to generate tests to cover targeted program branches. Typical test generation solution includes dynamic and static symbolic execution [9–11, 20, 46] and search-based software testing [5, 9, 18, 21, 26, 28, 29, 37]. While test coverage is an important metric for test completeness, readability and quality are also crucial for practical tests [14, 23, 38, 39]. Moreover, many tests are knowledge-driven. Thus, we design IntentionTest, an LLM-based solution. IntentionTest uses the general programming knowledge within the LLM and enhances the LLM with project-specific knowledge, which is complementary to coverage-based test generators.

LLM-based test generation. With the emergence of LLMs, researchers have made significant advances in code generation [44]. Considering test code is a special form of code, researchers have leveraged LLMs in software testing [2, 16, 19, 27, 30, 41, 42, 45, 53], considering the test generation problem as a translation problem from focal method to test code [15, 25, 33, 34, 47, 50, 52, 56]. One relevant work is ChatTester [56], which designs effective test-generating prompts and validates the tests based on a program analyzer and compiler. Different from ChatTester, IntentionTest is

a retrieval-and-edit solution. We further develop our contribution to how to effectively edit a referable test by discriminating the crucial project-specific information.

Additionally, IntUT [33] leverages generated test intentions to guide LLMs. A test intention in IntUT targets a specific branch within the focal method, specifying input parameters and the expected output when that branch is exercised. Consequently, IntUT is designed to maximize code coverage. Furthermore, expected outputs are derived directly from the focal method, implying that the focal method must return constant values. In contrast, our work introduces validation intentions, which are written by developers to reflect requirement fulfillment rather than coverage. Also, our approach imposes no constraints on the return behavior of the focal method.

8 Conclusion

In this work, we propose IntentionTest, which can generate project-specific tests with validation intention. Our solution is motivated by the empirical observation that mature software projects possess abundant code assets that can guide the generation of new tests. IntentionTest captures the project-specific knowledge by retrieving referable tests and collecting crucial facts, and derives practical tests using LLMs. We extensively evaluate IntentionTest on 3,680 tests from 12 projects. The results show that, compared to the state-of-the-art approaches, IntentionTest generates far more practical tests with better alignment to developers' validation intention.

9 Data Availability

Our source code and experimental data are available at [4].

References

[1] 2021. IEEE/ISO/IEC International Standard for Software and systems engineering–Software testing–Part 3:Test documentation. *ISO/IEC/IEEE 29119-3:2021(E)* (2021), 1–98.

[2] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 185–196.

[3] Anonymous. 2025. Anonymous video for IntentionTest tool. https://youtu.be/i1qMPqb993A.

[4] Anonymous. 2026. Anonymous website for IntentionTest. https://sites.google.com/view/domain-specific-tester/home.

[5] Andrea Arcuri and Xin Yao. 2008. Search based software testing of object-oriented containers. *Information Sciences* 178, 15 (2008), 3075–3095.

[6] Spark authors. 2023. Spark - a tiny web framework for Java 8. https://github.com/perwendel/spark.

[7] Tobias Baum and Kurt Schneider. 2016. On the need for a new generation of code review tools. In *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17*. Springer, 301–308.

[8] blade authors. 2025. Lightning fast and elegant mvc framework for Java8. https://github.com/lets-blade/blade.

[9] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2017. Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 90–101.

[10] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2018. SUSHI: a test generator for programs with complex structured inputs. In 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion).

[11] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[12] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. 2014. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 55–66.

[13] cron-utils authors. 2025. Cron utils for parsing, validations and human readable descriptions as well as date/time interoperability. https://github.com/jmrozanec/cron-utils.

[14] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 107–118.

[15] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. 2022. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*. 2130–2141.

[16] Chunhao Dong, Yanjie Jiang, Yuxia Zhang, Yang Zhang, and Liu Hui. 2025. ChatGPT-Based Test Generation for Refactoring Engines Enhanced by Feature Analysis on Examples . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 746–746. doi:10.1109/ICSE55347.2025.00210

[17] Emad Fallahzadeh, Amir Hossein Bavand, and Peter C Rigby. 2023. Accelerating Continuous Integration with Parallel Batch Testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 55–67.

[18] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[19] Shuzheng Gao, Chaozheng Wang, Cuiyun Gao, Xiaoqian Jiao, Chun Yong Chong, Shan Gao, and Michael Lyu. 2025. The Prompt Alchemist: Automated LLM-Tailored Prompt Optimization for Test Case Generation. arXiv:2501.01329

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223.

[21] Javier Godoy, Juan Pablo Galeotti, Diego Garbervetsky, and Sebastián Uchitel. 2021. Enabledness-based testing of object protocols. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–36.

[22] Larisa Gota, Dan Gota, and Liviu Miclea. 2020. Continuous Integration in Automation Testing. In *2020 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*. IEEE, 1–6.

[23] Giovanni Grano, Simone Scalabrino, Harald C. Gall, and Rocco Oliveto. 2018. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the 26th Conference on Program Comprehension*. 348–351.

[24] itext-java authors. 2025. iText for Java represents the next level of SDKs for developers that want to take advantage of the benefits PDF can bring. https://github.com/itext/itext-java.

[25] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[26] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[27] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 14–26.

[28] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-based seed object synthesis for search-based unit testing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1068–1080.

[29] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. 2020. Recovering fitness gradients for interprocedural Boolean flags in search-based testing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 440–451.

[30] Simone Mezzaro, Alessio Gambi, and Gordon Fraser. 2024. An empirical study on how large language models impact software testing learning. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 555–564.

[31] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. 2023. ClarifyGPT: Empowering LLM-based Code Generation with Intention Clarification. *arXiv preprint arXiv:2310.10996* (2023).

[32] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2332–2354.

[33] Zifan Nan, Zhaoqiang Guo, Kui Liu, and Xin Xia. 2025. Test Intention Guided LLM-Based Unit Test Generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1026–1038.

[34] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. 2023. Learning deep semantics for test completion. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2111–2123.

[35] ofdrw authors. 2025. OFD Reader & Writer. https://github.com/ofdrw/ofdrw.

[36] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2024. An Empirical Study of the Non-determinism of ChatGPT in Code Generation. *ACM Trans. Softw. Eng. Methodol.* (2024). doi:10.1145/3697010

[37] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[38] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2016. 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*. 5–14.

[39] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2016. Automatic test case generation: what if test code quality matters?. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 130–141.

[40] Martin Pawelczyk, Seth Neel, and Himabindu Lakkaraju. 2023. In-context unlearning: Language models as few shot unlearners. *arXiv preprint arXiv:2310.07579* (2023).

[41] Binhang Qi, Yun Lin, Xinyi Weng, Chenyan Liu, Hailong Sun, Gordon Fraser, and Jin Song Dong. 2026. Generalizing Test Cases for Comprehensive Test Scenario Coverage. *Proceedings of the ACM on Software Engineering* 3, FSE (2026). doi:10.1145/3808216

[42] Binhang Qi, Hailong Sun, Wei Yuan, Hongyu Zhang, and Xiangxin Meng. 2021. Dreamloc: A deep relevance matching-based framework for bug localization. *IEEE Transactions on Reliability* 71, 1 (2021), 235–249.

[43] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[44] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[45] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[46] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272.

[47] Jiho Shin, Sepehr Hashtroudi, Hadi Hemmati, and Song Wang. 2024. Domain Adaptation for Code Model-Based Unit Test Case Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1211–1222.

[48] Shota Takashiro, Takeshi Kojima, Andrew Gambardella, Qi Cao, Yusuke Iwasawa, and Yutaka Matsuo. 2024. Answer When Needed, Forget When Not: Language Models Pretend to Forget via In-Context Knowledge Unlearning. *arXiv preprint arXiv:2410.00382* (2024).

[49] truth authors. 2025. Fluent assertions for Java and Android. https://github.com/google/truth.

[50] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).

[51] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 1069–1088.

[52] Jin Wen, Qiang Hu, Yuejun Guo, Maxime Cordy, and Yves Le Traon. 2025. Variable Renaming-Based Adversarial Test Generation for Code Model: Benchmark and Enhancement. *ACM Transactions on Software Engineering and Methodology* (2025).

[53] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[54] Chen Yang, Junjie Chen, Bin Lin, Ziqi Wang, and Jianyi Zhou. 2025. Advancing Code Coverage: Incorporating Program Analysis with Large Language Models. *ACM Trans. Softw. Eng. Methodol.* (July 2025). doi:10.1145/3748505 Just Accepted.

[55] yavi authors. 2025. A lambda based type safe validation for Java. https://github.com/making/yavi.

[56] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.* 1, FSE, Article 76 (jul 2024), 24 pages. doi:10.1145/3660783

[57] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, Junda He, and David Lo. 2023. Generation-based code review automation: how far are we? In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 215–226.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009