2026 IEEE 37th International Symposium on Software Reliability Engineering (ISSRE)

TENET: One Step Toward Test-Driven Development for Repository-Level Code Generation

Yiran Hu, Shanchao Liang, Nan Jiang†, Yi Wu, and Lin Tan
Purdue University
{hu954, liang422, wu1827, lintan}@purdue.edu, †nan.nathan.jiang@gmail.com

***Abstract*—Test-Driven Development (TDD) is a widely adopted practice that requires developers to create and execute tests alongside implementation. With recent advances in Large Language Models (LLMs), developers can shift from manually writing the code to defining tests as executable specifications and delegating code synthesis to AI agents. However, enabling repository-level TDD under developer-written tests is challenging, requiring: (1) specification enhancement: identifying a concise yet representative test subset from large suites with rich task semantics; (2) retrieval augmentation: using tests to guide reasoning and context retrieval; and (3) test-driven refinement: interpreting test feedback for iterative improvement. We propose TENET, an agentic framework for repository-level code generation under the TDD paradigm. TENET includes: (1) a *test harness mechanism* that selects a concise test suite considering both the caller diversity and invocation proximity; (2) a *tailored agent toolset* for efficient retrieval and debugging; and (3) a *reflection-based refinement workflow* that iteratively analyzes failures and updates implementations. TENET consistently outperforms the strongest baselines across backbones, achieving 69.08% and 81.77% Pass@1 on REPOCOD and RepoEval with Claude Sonnet 4, improving by 9.49 and 2.17 percentage points, respectively. Additionally, we present the first systematic study of how test suite characteristics influence LLM agent performance in TDD settings.**

***Index Terms*—Test-Driven Development, Repository-Level Code Generation, Large Language Models, AI Agent.**

I. INTRODUCTION

Test-Driven Development is a widely adopted practice in software engineering that tightly couples the testing and implementation processes [1]: developers typically start by writing test cases that specify the desired behavior or capture potential failure scenarios, then incrementally implement and refine code to satisfy these tests. Extensive studies have demonstrated that TDD improves implementation accuracy, code quality, and design clarity [2–11].

In the era of AI-assisted coding [12], developers increasingly delegate code writing to large language models by providing high-level intentions. However, such descriptions often fail to capture detailed functionality, making it challenging for LLMs to implement correctly and sometimes even amplifying the ambiguity, especially with the repository-level context. This makes TDD even more crucial, as it systematically specifies and validates requirements through executable test cases to improve functional correctness. Recent industry practices further highlight the effectiveness of TDD in agentic coding workflows [13, 14].

Fig. 1 shows how an LLM agent generates the target function for `scikit-learn_304` from REPOCOD [15]. In the standard setting, the agent is given ① the task description, ② repository context, and ③ function specification. Based on these, it assumes that `y_prob.shape[1]` (the size of distributions) is always greater than 1, since classification should involve at least two classes, and produces the implementation in ⑤. However, `scikit-learn` also requires supporting singleton probability representations for binary classification (e.g., `[0.9]` instead of `[0.1,0.9]`), which cannot be inferred from ①–③ and the implementation in ⑤ fails on such inputs. In contrast, the TDD setting additionally provides ④ test cases that include such usage examples (the gray lines in ④) for specification enhancement, guiding the agent to generate the correct implementation in ⑥.

While this highlights TDD as a promising setting, existing studies on automated code generation in TDD are largely limited to standalone function generation [2–4, 16, 17]. Automating TDD in repository-level code generation remains underexplored. Although the ultimate goal of AI-driven TDD involves authoring new tests and the full "test-implement-refactor" cycle, a crucial step forward is enabling agents to reliably utilize and satisfy **existing developer-written** tests. This focused setting introduces several challenges.

First, unlike standalone functions, where tests directly invoke the target, software repositories contain large test suites where tests reach the target function through long and indirect call chains. How to strategically leverage these tests for **specification enhancement** to guide LLMs under computational budgets remains an open question. Second, relevant context is often scattered across the repository, and implementations must follow repository-specific structures, imports, and conventions that are invisible at function granularity. This poses a key challenge for **retrieval augmentation**: effectively leveraging test context and signals to guide agents toward effective reasoning and retrieval. Finally, test execution feedback is often sparse and noisy, and failures may arise from interactions far from the generated function, making it difficult for models to precisely interpret for **code refinement**.

To tackle these challenges, we introduce TENET, an agentic framework for repository-level code generation focusing on how to effectively utilize developer-written tests, retrieve informative context, and leverage test feedback for code refinement. The framework features three components. First, given developer-written test cases, TENET employs a ***test harness***

© 2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

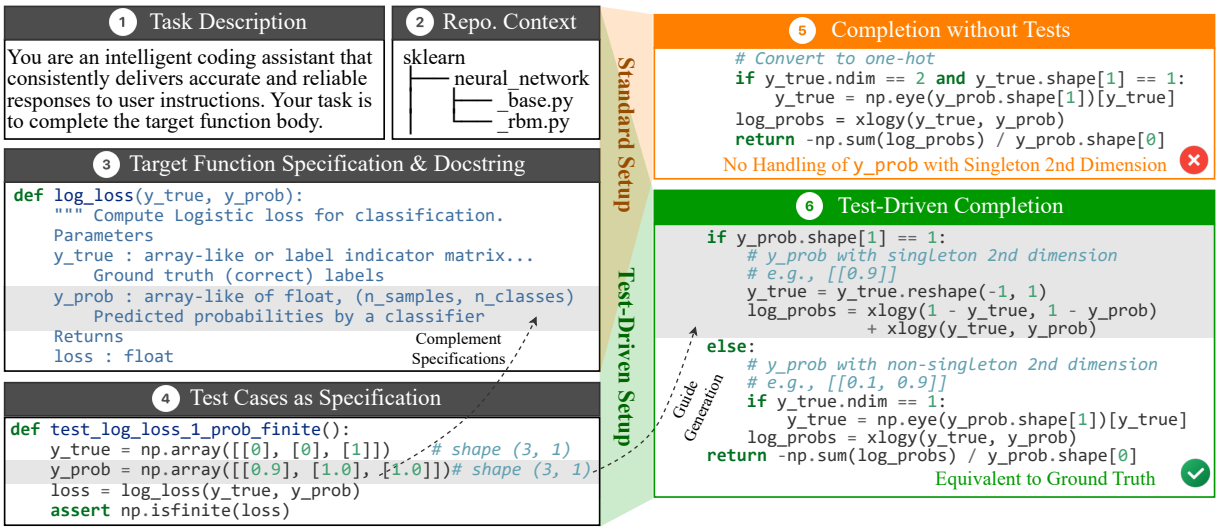


Fig. 1: Examples of repository-level code generation under standard and test-driven setups.

mechanism (THM) that utilizes dynamic analysis to select a subset of tests that invoke target functions from distinct caller functions in the call stack to maximize coverage of diverse usage scenarios. Second, it leverages a ***tailored agent toolset*** for repository-level code generation, enabling efficient discovery of diverse usage patterns of the target function and supporting interactive debugging. Third, it implements a ***reflection-based refinement workflow (RRW)*** that iteratively improves generated code via failure analysis, contextual replenishment, and execution-based debugging. Together, these components allow TENET to generate more accurate code in complex real-world repositories. The core contributions are summarized as follows.

- We propose an effective test harness mechanism for efficient test-driven supervision, using dynamic analysis to select a concise subset of tests for specification enhancement.
- We design a tailored agent toolset that supports LLM agents to efficiently utilize test signals and perform context retrieval with interactive debugging.
- We build a reflection-based refinement workflow for test-aware debugging, enabling LLM agents to precisely interpret test feedback for code refinement.
- We develop TENET, a test-driven agentic framework that integrates the above components for repository-level code generation. It achieves 69.08% and 81.77% Pass@1 on REPOCOD and RepoEval, outperforming the strongest agentic baselines by 9.49 and 2.17 percentage points (pp), respectively. These margins further widen to 20.10 and 6.97 pp under the DeepSeek-V3 backbone.
- We conduct the first systematic study of test-driven code generation using LLM agents at the repository level. The key findings include:
  - Too many tests do not necessarily lead to superior results; a moderate number, typically three to five representative tests, often yields optimal performance (Section V-C).
  - Test suites considering both caller diversity and invocation proximity provide the most effective guidance for LLM agent repository-level code generation (Section V-D).
  - While incorporating test signals in both context retrieval and refinement consistently improves the performance, it comes with higher token consumption. We should balance the trade-off between accuracy and efficiency (Section V-E).

Our implementation and results are available at https://github.com/lt-asset/tenet/.

II. Related Work

A. Test-Driven Development

A typical Test-Driven Development (TDD) workflow follows a "test–implement–refactor" cycle: developers begin with test cases that specify desired functionality, write the minimal code required to pass the tests, and refactor the implementation while ensuring tests continue to pass. Prior work has demonstrated the effectiveness of TDD in improving software quality [5–9, 11] and has further explored its application to LLM-based code generation, primarily on standalone function generation benchmarks [2–4, 16, 17]. Under this setting, models do not need to navigate complex file dependencies or isolate the specific subset of tests relevant to a change.

In contrast, TENET takes a step towards repository-level TDD. Our setting requires the agent to select, utilize, and satisfy the **existing developer-written** test cases in the repository[18, 19]. It introduces new challenges in test selection for specification enhancement and test-guided retrieval that standalone baselines do not address.

B. Repository-Level Code Generation

Many code generation benchmarks focus on generating isolated code units [20–22], which differ from real-world software development where implementations must integrate with large repositories and satisfy project-specific dependencies. To better reflect such scenarios, repository-level code generation benchmarks have been proposed [15, 23–29], requiring models

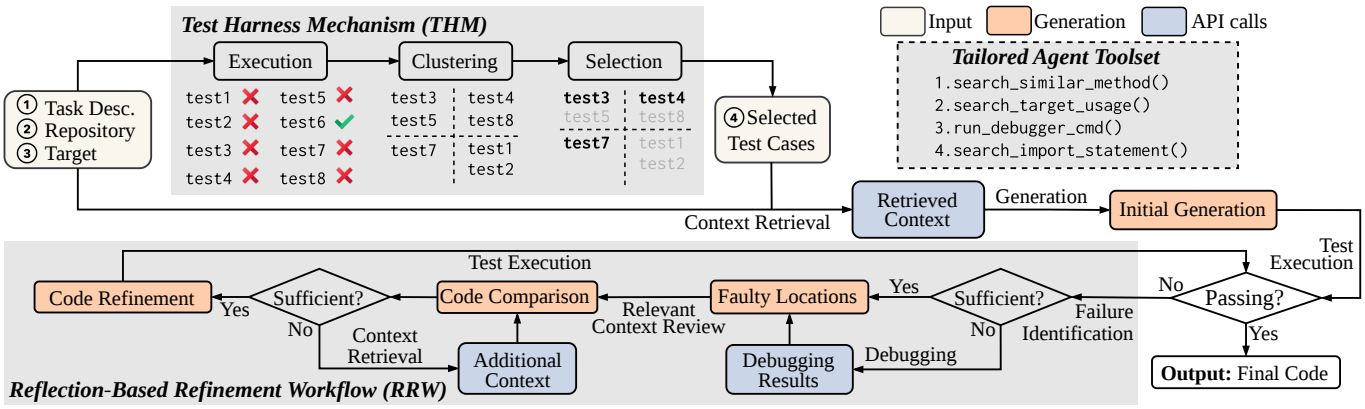


Fig. 2: TENET workflow.

to implement new functionalities within complex codebases while maintaining consistency.

Existing approaches for repository-level code generation with LLM can be categorized into non-agentic and agentic methods. Non-agentic approaches mainly follow two paradigms: Retrieval-Augmented Generation (RAG), which retrieves relevant repository context using semantic or structural strategies [23, 30–39]; and feedback-driven refinement, which improves generated code using external signals such as compiler errors [40] or symbolic planning [41]. Agentic approaches formulate repository-level code generation as a multi-step decision-making process, where the model interacts with external tools to explore the repository, modify code, and validate changes [42–45].

C. Large Language Model Agents

LLM agents leverage LLMs as central reasoning engines to decompose complex problems, formulate multi-step plans, and interact with external environments through tool invocation. Prior work explores diverse agent designs for software engineering tasks. Approaches including AutoCodeRover [46], SpecRover [45], as well as CodeAgent [42], are designed for specific scenarios, such as issue fixing or repository-level code generation. These methods typically rely on predefined workflows and toolsets to guide program analysis and code modification. General-purpose software engineering agents, including SWE-Agent [44], CodeAct [47], and Live-SWE-Agent [48], can solve real-world GitHub issues by equipping LLMs with tools for code navigation, editing, and test execution, with Live-SWE-Agent further enabling agents to dynamically construct their own toolsets on the fly.

However, when generating code in a test-driven manner, they lack mechanisms to select informative tests and to effectively leverage execution feedback for refinement. TENET addresses this by using THM to select diverse and informative test cases for better specification, equipping LLMs with a powerful toolset for retrieval and interactive debugging, and adopting RRW for structured and effective code refinement.

III. APPROACH: TENET

Fig. 2 illustrates the workflow of TENET, which consists of three major components: (1) the *test harness mechanism (THM)*, (2) the *tailored agent toolset*, and (3) the *reflection-based refinement workflow (RRW)*. The THM is a dynamic-analysis preprocessing stage that operates *outside* the agent: it first selects a small yet effective test subset from the repository's complete test suite. This subset is provided as additional input ④, alongside the ① task description, ② repository context, and ③ target function specification. Throughout the subsequent workflow, the agent can access and execute only the selected tests, while all remaining tests are held out from the agent. Guided by the test subset, TENET explores the repository using the tailored agent toolset, which supports efficient retrieval and interactive debugging. After gathering sufficient context, it generates a solution and validates it against the selected tests. If any test fails, TENET enters the RRW to repair the generated code.

During the RRW, the agent assesses whether the current information provides adequate support for a correct refinement, including fault localization and issue understanding. If not, the RRW prompts the agent to retrieve additional context and use interactive debugging to gather further insights. This loop continues until the agent deems the evidence sufficient for a fix, at which point it generates a candidate refinement. The updated code is then validated using the THM-selected tests. TENET stays in the RRW until all selected tests pass or a maximum number of attempts is reached.

A. Test Harness Mechanism

Test cases can expose diverse usage scenarios of the target function, serving as a form of specification enhancement beyond the function signature and docstring. However, dumping the entire test suite to the LLM is infeasible due to two reasons.

- **Effectiveness**. Providing massive test cases can overwhelm the LLM with long, complex prompts, increasing the cognitive load of the model [2, 49].
- **Efficiency**. Such massive test suites and long prompts inevitably incur higher generation latency and execution workload, reducing the efficiency of the agent pipeline [50].

To identify a concise and effective test subset from the entire repository test suite, TENET employs an effective *THM* that first executes the full test suite against the unimplemented target function to collect failing cases. These failing cases

are then clustered by the *caller* function that directly invokes the target function in their call stack. The intuition is that different caller functions are likely to expose distinct usage patterns and test diverse aspects of the target function's logic, thereby providing complementary test coverage. For the example shown in Fig. 1, the test function with call chain `test_log_loss_1_prob_finite` → `log_loss` tests the target function under the binary classification scenario and covers the `if` branch, while another cluster of test functions, with call chain ... → `_backprop` → `log_loss`, all test the target function by covering the `else` branch.

From these clusters, TENET selects at most T test cases. The selection further prioritizes tests with the shortest call chain from the entry test function to the target function. The rationale is that tests with shorter invocation paths may expose cleaner execution contexts and more localized failure signals. In contrast, tests with deeper invocation paths often provide longer and more complex feedback, indicating complicated execution paths and error propagation. Selecting tests with shorter call chains may expose the agent to more concise and focused feedback, thereby simplifying the refinement process and reducing the overall task difficulty.

In practice, the process first attempts to pick one representative test case from each cluster with the shortest call chain. If the number of clusters is greater than or equal to T, the top T clusters are chosen, with one test case selected from each. If the number of clusters is fewer than T, additional test cases with the shortest call chains are selected until the budget of T is reached. This strategy ensures that the final subset of test cases remains both diverse across failure patterns and closely tied to the target function's behavior.

We emphasize that the full-suite execution above is performed by the THM alone for candidate discovery; the subsequent agent only has access to the selected tests. We set $T = 3$ for the algorithm described above based on preliminary experiments on the sphinx project from REPOCOD (Section V-C). We study alternative test selection strategies in Section V-D.

B. Tailored Agent Toolset

TENET provides a tailored toolset that extends the abstract-syntax-tree (AST)-based toolset of SpecRover [45] for structural context retrieval and interactive debugging. The API toolsets of existing agents can be roughly divided into two categories: AST-based, such as AutoCodeRover and SpecRover, and terminal-command based, such as SWE-Agent [44] and OpenHands [43]. While AST-based interfaces allow structural navigation and terminal-command based interfaces offer more flexibility, they still face several limitations when generating functions under repository-level context.

First, RAG techniques can substantially improve generation accuracy by providing relevant code examples. However, existing LLM agents have not incorporated semantic retrieval as an API beyond basic repository navigation, which usually requires multiple attempts to locate the desired context. Second, understanding the use cases is crucial for accurate code generation. Yet, existing LLM agents rely on terminal commands to find the usage of a certain function based on string matching, which is inefficient and error-prone. Third, although interactive debugging plays an important role in refining code [51], existing agents treat it as an end-to-end process, lacking support for fine-grained interactive debugging that enables stepwise evidence collection and fault diagnosis.

To address these, we extend SpecRover's AST-based toolset with four new APIs:

- **`search_similar_method(n)`** retrieves the top-`n` methods most relevant to the signature and docstring of the target function, ranked by BM25 similarity [15, 42]. This API enables the agent to collect context as references efficiently.
- **`search_target_usage(n)`** retrieves up to n usage examples of the target function via AST analysis. Unlike keyword-matching commands that require multiple indirect queries and often return noisy snippets, this API provides usage contexts in a single step, making it easier for the agent to understand how the target function is invoked.
- **`run_debugger_cmd(cmd)`** executes terminal commands within a container session, including interactive debugger commands (e.g., `pdb`) for step-by-step execution, variable inspection, and stack frame traversal.
- **`search_import_statement(f)`** retrieves all top-level import statements in the specified file `f`. This enables the agent to analyze dependencies and disambiguate call paths in cross-file analysis.

Together, these APIs address the identified limitations by allowing TENET to efficiently retrieve relevant context and conduct structured interactive debugging.

C. Reflection-Based Refinement Workflow

With the THM and the tailored toolset, TENET first generates a code snippet which is then validated against the chosen test cases. If any THM-selected test fails, TENET enters the RRW to revise the code snippet iteratively. Throughout the RRW, generation and refinement are validated *exclusively* against the THM-selected tests. Unlike refinement in the setting of self-contained functions, where the LLM only needs to review and revise its own faulty implementation [52–54], the generation of incorrect code within a repository could be due to more complex reasons. For example, LLM can get "lost in the middle" due to long trajectories [49], overlooking important context such as the usage of repository-specific functions.

To address such issues, TENET employs a structured and bounded *RRW* rather than relying on unguided self-reflection. Each refinement attempt follows three stages: (1) **Failure Identification.** The LLM identifies potentially faulty locations from the current trajectory and invokes debugger commands only when additional execution evidence is needed. (2) **Context Review.** The LLM is then prompted to identify relevant code snippets (e.g., implementations of similar functionality) based on the current trajectory to avoid redundant tool calling. When such snippets are available, the LLM is guided to explain their implementations, such as the handling of edge cases or the usage of specific functions, and compare them with the faulty implementations to extract insights for bug

fixing. (3) **Code Refinement.** The LLM determines whether the available evidence is sufficient for refinement. If so, it formulates and applies a concrete repair strategy; otherwise, it selectively gathers additional context or debugging signals before revising the solution. The RRW terminates when either all THM-selected tests pass or the refinement budget is exhausted, preventing indefinite oscillation among failing solutions.

IV. Experimental Setup

A. Research Questions

To assess the effectiveness of TENET and to gain deeper insights into repository-level code generation under the TDD setting, we formulate the following research questions (RQs).

- **RQ1:** How effective is TENET compared to other repository-level code generation baselines?
- **RQ2:** How does each component of TENET contribute to its overall effectiveness?
- **RQ3:** What effect does the quantity of test cases in TENET have on code generation performance?
- **RQ4:** How do different test selection strategies affect the code generation pass rate?
- **RQ5:** What is the impact of using test cases at different stages of TENET on performance?

B. Benchmarks

We evaluate our approach on two repository-level code generation benchmarks, REPOCOD [15] and the function-level tasks of RepoEval [23]. Both benchmarks preserve full repository context and executable test suites, comprising 980 and 373 function generation tasks, respectively. Each task requires synthesizing a missing target function such that the completed implementation passes the associated tests. Although both benchmarks fall under repository-level code generation, they differ in both the scope of required code modifications and the size of test suites. REPOCOD tasks typically involve synthesizing more substantial code, with an average of 38.18 edited lines of code (LOC) and 334.42 code-edit tokens[1]. Each task is evaluated against approximately 68 tests[2] to determine Pass@1. In contrast, RepoEval tasks generally require more concise implementations, averaging 9.78 edited LOC and 84.92 code-edit tokens, with approximately 15 evaluation tests per task. This distinction enables evaluating TENET across varying implementation scopes. For each task, we execute all agentic approaches in isolated Docker environments with complete dependencies to ensure consistent and reproducible evaluation.

A related but distinct line of work focuses on automated issue fixing, exemplified by SWE-Bench Verified [55–57]. In this setting, the primary objective is software maintenance: agents interpret issue descriptions, localize faulty code, and generate minimal patches to repair existing implementations. Prior analysis shows that 87.20% of SWE-Bench Verified tasks correspond to bug fixing, while only 8.60% involve feature implementation [58]. This distribution reflects a fundamental difference in task scope. Unlike issue-fixing benchmarks that emphasize fault localization and patch generation, the setting we consider requires synthesizing new functionality under test-driven constraints, introducing distinct challenges in test selection, context understanding, and test-guided refinement.

[1]The average number of code-edit tokens per task is computed using a fixed byte-pair encoding tokenizer from the `tiktoken` library.

[2]This count does not distinguish test functions executed with different inputs. When each input variant is counted as a separate test, the average increases to 313 [15].

C. Baselines

Across all RQs, we use DeepSeek-V3 [59] as the primary backbone for its cost efficiency. In RQ1, we compare TENET with four open-source baselines covering both non-agentic and agentic paradigms, and further evaluate all methods with Claude Sonnet 4 [60] as a high-performance reference. All baselines receive the task description and full repository context, including the complete test suites. Agentic baselines may freely inspect and execute any repository test. In contrast, TENET exposes only the THM-selected tests to the downstream agent; all other tests are withheld during generation and refinement and used only for final evaluation. We configure TENET with up to three THM-selected tests, 15 pre-generation conversation rounds, five RRW refinement attempts, and 15 conversation rounds per refinement attempt for debugging and context retrieval. For all baselines and configurations, we set the model temperature to 0 and sample one patch per task.

1) **RepoCoder** adopts an iterative retrieval–generation framework [23]. We follow the original configuration on RepoEval. For REPOCOD, we use a 12,288-token retrieval window, retrieve up to 30 snippets per query, and cap the maximum completions at 4,096 tokens.

2) **SpecRover** [45] is a multi-agent framework coordinating specialized agents for retrieval, generation, testing, and reflection. We follow the same configuration in the paper.

3) **SWE-Agent** equips LLMs with shell commands and custom actions [44] for general software engineering tasks. We use a maximum of 50 conversation rounds for each task.

4) **OpenHands** is an open-source platform for developing software engineering agents [43]. We employ the default CodeAct Agent [47], which supports shell command execution, file reading, and file editing. OpenHands follows the same conversation rounds as SWE-Agent.

The round budgets for SWE-Agent and OpenHands are relatively higher, as they support only one tool invocation per response, while TENET can perform multiple API calls in a single round. This adjustment balances comparable tool-calling capacity and cost efficiency.

V. Evaluation Results

A. RQ1: Baseline Comparison

We report the performance of TENET and baseline approaches on REPOCOD and RepoEval under two backbones in Table I, including Pass@1 [20], average token consumption, average monetary cost, and Cost-of-Pass (CoP) [61]. CoP is defined as the expected monetary cost to obtain one correct solution for a problem, computed as the ratio between the

TABLE I: Comparison of Pass@1, token consumption, monetary cost, and Cost-of-Pass (CoP) on REPOCOD and RepoEval under Claude Sonnet 4 and DeepSeek-V3.

Backbone	Approach	REPOCOD					RepoEval				
		Pass@1 (%)↑	Input ↓	Output ↓	Cost ($)↓	CoP ↓	Pass@1 (%)↑	Input ↓	Output ↓	Cost ($)↓	CoP ↓
Sonnet 4	RepoCoder	26.22	14,225	787	$0.03	0.11	53.08	6,048	306	$0.01	0.02
	SpecRover	33.33	95,884	6,609	$0.35	1.05	72.12	52,317	4,643	$0.20	0.28
	OpenHands	59.18	1,119,149	8,988	$0.64	1.08	79.60	471,103	4,434	$0.30	0.38
	SWE-Agent	59.59	597,507	1,252	$0.45	0.76	67.02	344,374	1,242	$0.30	0.45
	TENET	**69.08**	194,932	6,560	$0.53	0.76	**81.77**	111,934	3,790	$0.28	0.34
DeepSeek-V3	RepoCoder	19.80	12,477	511	$0.004	0.018	48.26	12,830	222	$0.003	0.007
	SpecRover	18.24	61,378	5,456	$0.023	0.124	46.11	46,603	4,256	$0.017	0.038
	OpenHands	29.08	353,648	6,608	$0.051	0.176	43.16	176,073	2,210	$0.025	0.059
	SWE-Agent	27.04	386,932	736	$0.049	0.181	35.66	70,540	533	$0.007	0.020
	TENET	**49.18**	147,968	4,645	$0.026	0.052	**55.23**	82,473	2,967	$0.017	0.031

average monetary cost per task and the Pass@1 of an approach. Lower CoP indicates a more favorable accuracy-cost trade-off.

Accuracy Gains. TENET achieves the highest Pass@1 across *both* benchmarks under *both* backbones. Under Sonnet 4, it reaches 69.08% and 81.77% Pass@1, surpassing the strongest baselines by 9.49 and 2.17 pp, respectively. Under DeepSeek-V3, TENET attains 49.18% and 55.23%, where its margin over the strongest baselines *widens* to 20.10 and 6.97 pp. This demonstrates the effectiveness of our design.

Cost Efficiency. TENET achieves a favorable accuracy–cost tradeoff under both backbones.

1) *Token Consumption.* Agentic baselines incur substantially higher token usage compared with RepoCoder. On REPOCOD under Sonnet 4, OpenHands and SWE-Agent consume 1,119,149 and 597,507 input tokens per task on average, corresponding to 574.12% and 306.52% of TENET, respectively. A similar trend is observed on RepoEval (420.88% and 250.46% of TENET's input tokens) and under DeepSeek-V3 (239% and 261% on REPOCOD). This overhead primarily stems from two factors: (1) lengthy system prompts that encode detailed execution policies, security constraints, and multi-step workflows; and (2) reliance on terminal-level commands (e.g., `grep`, `find`) that require many sequential, fragmented steps for code retrieval. These interactions cause trajectories to grow rapidly, resulting in a dramatic increase in token consumption.

2) *Monetary Cost.* Under Sonnet 4 on REPOCOD, TENET attains a higher Pass@1 while maintaining a comparable CoP (0.76) to SWE-Agent, which achieves the strongest Pass@1 and CoP among agentic baselines; on RepoEval, it achieves the highest Pass@1 with a competitive CoP of 0.34, only slightly higher than SpecRover, while outperforming it by 9.65 pp in Pass@1. The advantage is even clearer under the consistent DeepSeek-V3 backbone: TENET attains the *lowest CoP among all agentic approaches* on REPOCOD, and a competitive CoP on RepoEval (0.031), second only to SWE-Agent, which trails TENET by 19.57 pp in Pass@1.

Overall, TENET achieves high accuracy without sacrificing cost efficiency, and its advantage holds under different backbone models. While TENET excels, general-purpose agents exhibit competitive performance across benchmarks [55, 58, 62, 63], reflecting their strong generality.

TABLE II: Ablation study of TENET using DeepSeek-V3 on REPOCOD. Compared with TENET, red arrows indicate worse results and blue arrows indicate better results.

Variants	Pass@1 (%) ↑	Input ↓	Output ↓	API Call ↓
TENET$_{\text{-THM}}$	33.06↓ 16.12 pp	208,179↑ 40.69 %	5,547↑ 19.41 %	12.05↑ 45.53 %
TENET$_{\text{-APIs}}$	34.29↓ 14.89 pp	138,031↓ 6.72 %	6,358↑ 36.87 %	10.53↑ 27.17 %
TENET$_{\text{-RRW}}$	39.94↓ 9.24 pp	132,427↓ 10.50 %	4,008↓ 13.71 %	6.62↓ 20.05 %
TENET	49.18	147,968	4,645	8.28

B. RQ2: Contributions of TENET 's Components

To address RQ2, we first remove each component from the full TENET system to perform an ablation study on REPOCOD. Removing the THM (TENET$_{\text{-THM}}$) provides the agent with full target test suite instead of the selected subset; removing the tailored toolset (TENET$_{\text{-APIs}}$) limits the agent to the SpecRover toolset; and removing the RRW (TENET$_{\text{-RRW}}$) applies naive refinement given test feedback if test execution fails. Table II reports the Pass@1, average token consumptions (Columns "Input ↓" and "Output ↓"), and the number of API calls on REPOCOD. *Across all cases, removing any component results in a clear decline in Pass@1.*

1) Test Harness Mechanism: Removing THM causes the largest Pass@1 drop (16.12 pp). The token consumption also rises substantially with inputs by 40.69% and outputs by 19.41%, along with a 45.53% rise in API calls. This is mainly due to two factors. First, each REPOCOD task contains 68 test cases on average, far more than the curated suites from the THM. Feeding the model with the full suite introduces redundancy and noise, reducing accuracy. Second, handling the full suite requires analyzing more tests, which increases API calls and token usage.

Fig. 3 illustrates the task `seaborn_34` from REPOCOD to demonstrate the effectiveness of the THM. The full target test suite of `seaborn_34` contains 156 test cases for the target function `desaturate`, located in `seaborn/utils.py`. Without the THM, ⓪ the large test suite overwhelms the LLM, causing the agent to overlook critical test signals and generate ⑤ the incorrect code. Specifically, the generated function fails to satisfy the input validation requirements specified in `test_desaturation_prop`. It further fails the test `test_desaturate` with the following error message.

```
E AssertionError: assert (0.6784313725...9607843137256) ==
     (0.6784313725...9607843137255)
E At index 1 diff: 0.8470588235294119 != 0.8470588235294118
```

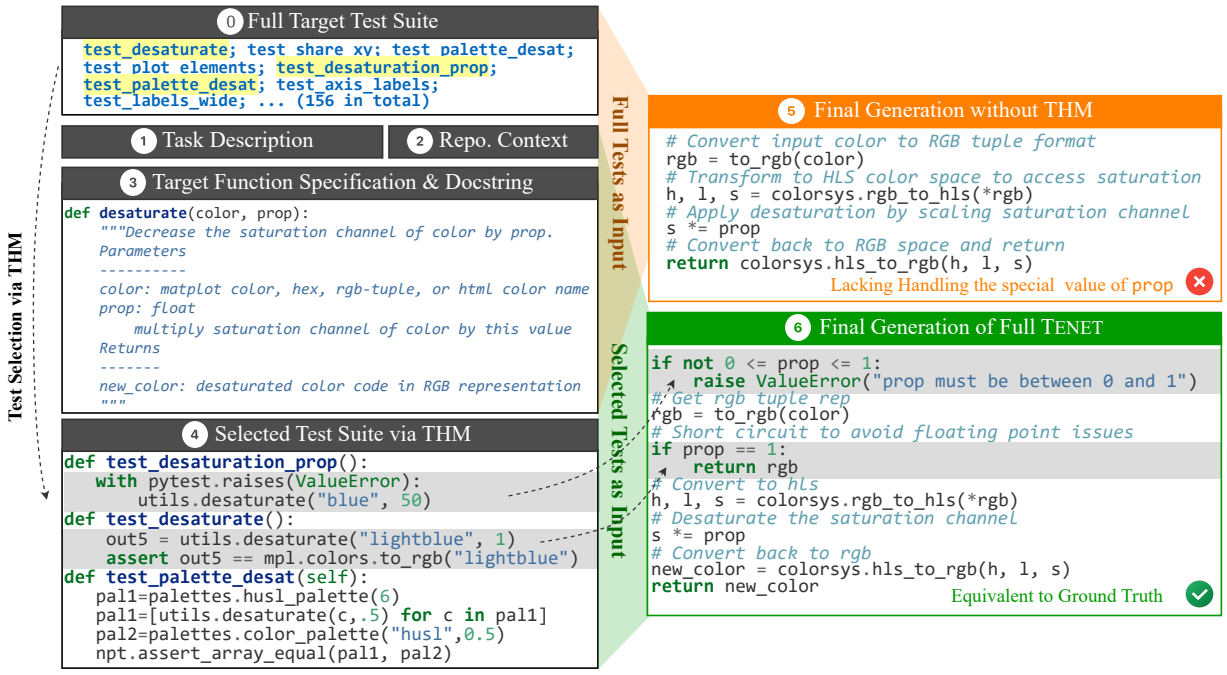


Fig. 3: Case study on task `seaborn_34` from REPOCOD, explaining how THM guides TENET toward correct code generation.

TABLE III: Effect of THM-selected tests on SWE-Agent using DeepSeek-V3 on REPOCOD.

Variants	Pass@1 (%) ↑	Input ↓	Output ↓	API Calls ↓
SWE-Agent	27.04	386,932	736	21.12
SWE-Agent$_{+THM}$	41.02 ↑ 13.98 pp	412,377 ↑ 6.58 %	916 ↑ 24.46 %	26.44 ↑ 25.19 %

The failure occurs because the incorrect code in ⑤ always performs an RGB→HLS→RGB round-trip, even for the boundary case `prop==1`, introducing small floating-point deviations that cause the test to fail. In contrast, THM selects three representative test cases without overwhelming the agent, and TENET uses two refinement rounds to generate the correct code in ⑥ that properly handles all edge cases.

We further examine whether THM-selected tests benefit other agentic baselines beyond TENET. Keeping the RQ1 baseline setting unchanged, we additionally provide SWE-Agent with the THM-selected tests for each task and instruct it to verify its implementation against them. Table III reports Pass@1, average token consumption, and API calls on REPOCOD. THM-selected tests improve SWE-Agent's Pass@1 by 13.98 pp, from 27.04% to 41.02%, at the cost of moderately higher token consumption and 25.19% more API calls. This gain is largely driven by SWE-Agent's original test-usage behavior: despite unrestricted access to the full test suite, it submits patches without any test verification in 28.98% of tasks. THM-selected tests, together with an explicit verification instruction, make testing a routine step in the workflow. **These results demonstrate that THM is an effective and transferable test-selection mechanism across agent designs.** However, SWE-Agent$_{+THM}$ still trails TENET by 8.16 pp, indicating that selected tests alone are insufficient; the tailored toolset and RRW are needed to fully exploit them.

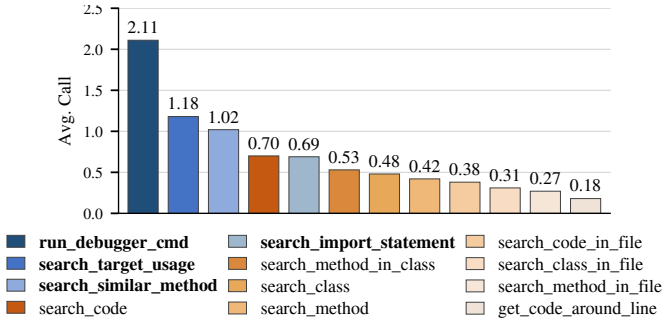


Fig. 4: Average API calls per task on REPOCOD (TENET, DeepSeek-V3).

2) Tailored Agent Toolset: Removing the tailored toolset reduces Pass@1 by 14.89 pp. Without TENET's specialized tools, such as retrieving semantically similar code, the agent must rely on less efficient APIs to analyze the context, which increases reasoning complexity. Though the input tokens are fewer, the more expensive output tokens and API calls increase substantially. Fig. 4 shows the average call frequency of different APIs[3]. Our four newly introduced APIs (colored in blue) rank among the top five. We observe that `run_debugger_cmd` dominates with the highest frequency (2.11), highlighting the frequent usage of interactive debugging in the RRW. The next most frequently used APIs are `search_target_usage` (1.18) and `search_similar_method` (1.02), suggesting that retrieving usage examples and similar methods are also favorable in the TDD setting. **These observations indicate that the model consistently favors our tailored toolset in the TDD setting, achieving efficient retrieval and improved**

[3]The APIs not highlighted in bold are inherited from SpecRover.

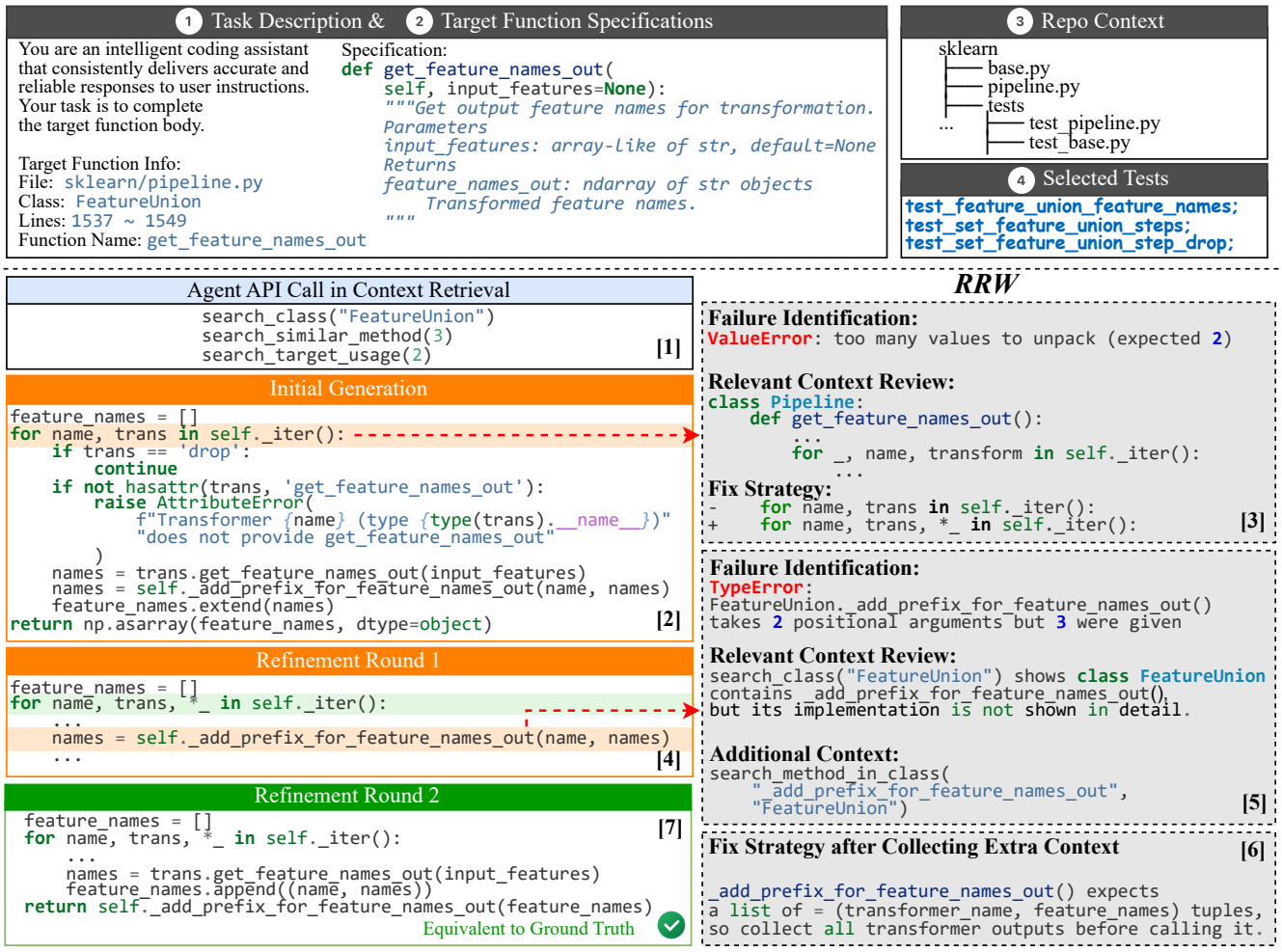


Fig. 5: Case study on task `scikit_49` in REPOCOD, explaining how the RRW contributes to the code refinement in TENET.

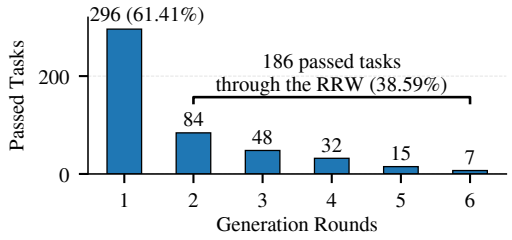


Fig. 6: Number of solved tasks per round (TENET, DeepSeek-V3).

performance with only a minor increase in input costs while reducing output costs and API overhead.

3) Reflection-Based Refinement Workflow: Removing the RRW leads to a decrease of 9.24 pp on Pass@1. We let TENET$_{\text{-RRW}}$ naively regenerate code without explicit reasoning about fault localization or code comparison. As a result, the performance drops while the token consumption and API calls slightly decrease. Shown in Fig. 6, solved tasks drop gradually over rounds, and out of 482 passed tasks in total, 296 (61.41%) are solved in the first attempt. The remaining 186 tasks (38.59%) are recovered through the RRW.

Fig. 5 illustrates the task `scikit_49` from REPOCOD to show how the RRW enables TENET to progressively correct generation errors and converge to the ground-truth solution. The task requires generating the `get_feature_names_out` method of the `FeatureUnion` class in `sklearn/pipeline.py`. TENET first invokes three APIs in `[1]` to retrieve the `FeatureUnion` class context, similar methods, and usage examples of the target function. Believing the collected context to be sufficient, TENET generates an initial implementation in `[2]`. However, test execution exposes a runtime error in `[3]`: *ValueError: too many values to unpack*. Guided by the RRW, TENET identifies the failure location, reviews the retrieved context, locates a similar loop structure in the reference implementation, and produces the first refinement in `[4]`.

Upon re-testing, a new error is triggered: *TypeError: _add_prefix_for_feature_names_out takes 2 positional arguments but 3 were given*. TENET quickly localizes the failure and recognizes that the current trajectory only contains the signature of `_add_prefix_for_feature_names_out`, but not its full body: the earlier `search_class("FeatureUnion")` call was truncated due to context-window constraints. This incomplete context causes the agent to invoke the method with an

TABLE IV: Pass@1 (%) of TENET with different test suite sizes on REPOCOD.

#Test	Sph.	Sea.	Fla.	Xar.	Sym.	More.	Data.	Skl.	Ast.	Pyl.	Plot.	Total
1	39.39	47.44	67.44	25.30	26.80	56.98	47.46	22.61	43.52	23.08	**43.42**	35.71
3	**54.55**	55.13	72.09	**42.17**	**34.02**	70.93	**62.71**	**46.18**	47.06	30.77	40.79	**49.18**
5	42.42	55.13	**79.07**	37.35	28.87	**80.23**	59.32	43.95	**50.59**	**34.62**	42.11	48.57
10	36.36	**57.69**	74.42	34.94	26.80	76.74	54.23	35.35	48.24	**34.62**	40.79	44.29
All	30.30	41.03	53.49	18.07	24.74	76.74	38.98	20.06	40.00	19.23	38.16	33.06

TABLE V: Pass@1 and test coverage of TENET (DeepSeek-V3) under different selection strategies on REPOCOD.

Strategies	Pass@1												Avg. Cov (%)
	Sea.	Fla.	Xar.	Sph.	Sym.	More.	Data.	Skl.	Ast.	Pyl.	Plot.	Total	
RS	44.87	**76.74**	32.53	42.42	28.87	68.60	44.07	12.10	34.12	19.23	36.84	32.86	70.94
SS	51.28	69.77	36.14	45.45	32.99	68.60	47.46	15.97	36.47	23.08	**43.42**	36.12	72.72
FRS	47.44	74.72	30.12	36.36	32.99	63.95	50.84	28.98	36.47	19.23	40.79	38.88	71.58
IPS	**55.13**	74.42	33.73	48.48	29.90	69.77	50.84	28.66	**47.06**	26.92	42.11	41.53	76.98
THM	**55.13**	72.09	**42.17**	**54.55**	**34.02**	**70.93**	**62.71**	**46.18**	**47.06**	**30.77**	40.79	**49.18**	**79.38**

incorrect argument structure. Therefore, TENET issues an additional query, `search_method_in_class("_add_prefix_for_feature_names_out", "FeatureUnion")`, to retrieve the complete method definition, as shown in `[5]`. With the correct usage clarified, TENET applies the fix in `[6]` and produces a final refinement in `[7]` that matches the ground-truth implementation.

In summary, despite increased token usage, the RRW consistently enhances the effectiveness of TENET, by iteratively identifying failure signals, reviewing context, and adaptively invoking APIs when necessary.

C. RQ3: The Impact of Test Suite Size

While incorporating more tests may offer richer feedback, it can also introduce redundant signals and increase overhead. To study this trade-off, we investigate how varying the number of selected tests influences the performance of TENET. As a tunable parameter, we set $T = 3$ based on preliminary results on the sphinx project (33 tasks) in REPOCOD. We use the default settings of TENET and only change the number of selected tests $T = \{1, 3, 5, 10, All\}$. Table IV reports Pass@1 across different test suite sizes on REPOCOD[4]. Overall, $T = 3$ delivers the highest overall Pass@1 (49.18%). $T = 5$ is optimal for four projects and ranks second overall. Moreover, performance generally declines as the test suite size increases. These results show that our finding on sphinx generalizes to the full benchmark. **Importantly, providing more tests does not necessarily improve outcomes, and a moderate number of tests provides consistent gains under the TDD setting.**

D. RQ4: The Impact of Test Selection Strategies

In the TDD setting, developers can design tests along multiple dimensions, such as test complexity, whether a test targets fine-grained functionality or spans multiple interacting components, and the form of validation such as assertions. These design choices may lead to tests with different features and feedback signals. Understanding such differences is important not only for agent design but also for informing how developers should construct tests in practice. To this end, we formulate RQ4 to compare THM with four test selection strategies that emphasize different test properties, and analyze how these strategies influence the generation performance.

*1) **Random Selection (RS)**:* RS is used as a baseline that uniformly samples a fixed number T of test cases from the full test suite, enabling a direct comparison with more informed selection strategies.

*2) **Simplicity-Based Selection (SS)**:* SS prioritizes test cases with low cyclomatic complexity, based on the intuition that simpler tests introduce fewer dependencies and provide clearer failure signals. We compute the cyclomatic complexity of each test function, rank them in ascending order, and select the top-T tests. In REPOCOD, 66.17% of tests per task have complexity below 4, indicating that most tests are relatively simple and supporting the practicality of using low-complexity tests to guide LLM-based code generation.

*3) **Failure-Revealing Selection (FRS)**:* FRS prefers tests that contain explicit assertions or exception checks. The intuition behind FRS is that tests with explicit validation logic are more likely to expose intended behaviors and produce actionable feedback when failures occur, compared to tests that simply execute code without verification, which may pass silently or fail with ambiguous signals. To implement FRS, we perform AST analysis to identify tests containing assertion-related constructs, including Python `assert`/`raise`, `pytest.raises`, `unittest` assertions, and their variants. We then randomly sample T tests from this subset to provide to the agent. In REPOCOD, 89% of tests contain assertion-related constructs, indicating that assertion-based tests are prevalent in practice. However, this also enlarges the candidate pool, making FRS behavior closer to RS and potentially reducing its distinctiveness.

*4) **Invocation-Proximity Selection (IPS)**:* IPS prioritizes test cases that invoke the target function through shorter call chains. For implementation, we measure the invocation depth of each test case following the THM procedure. We then rank tests in ascending order of depth and select the top-T tests.

Table V reports the complete Pass@1 and average test coverage of TENET on REPOCOD under different test selection strategies when $T = 3$. The average test coverage is computed based on the ground-truth target functions. All test selection strategies outperform the RS baseline, with improvements

[4]Repository abbreviations: Sph.=sphinx, Sea.=seaborn, Fla.=flask, Xar.=xarray, Sym.=sympy, More.=more-itertools, Data.=datasets, Skl.=scikit-learn, Ast.=astropy, Pyl.=pylint, Plot=plotly.py.

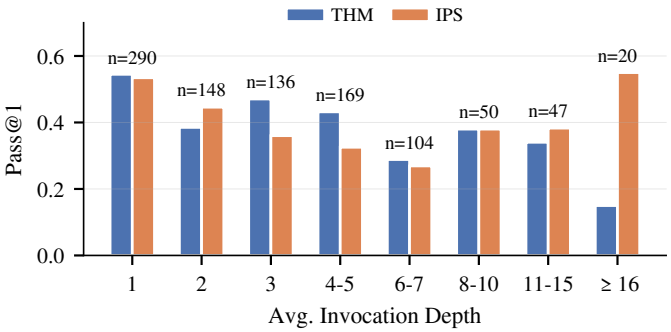


Fig. 7: Pass@1 performance of THM and IPS across tasks grouped by average invocation depth in the full test suite.

in Pass@1 generally aligning with higher coverage. Among them, TENET's default THM strategy, which emphasizes both invocation proximity and caller diversity, achieves the best results (49.18% Pass@1 and 79.38% coverage). This advantage can be attributed to the complementary roles of these two factors. Invocation proximity increases the likelihood that selected tests are semantically relevant to the target function, reducing the reasoning burden to relate the test behaviors with the target implementations. Meanwhile, caller diversity exposes the agent to multiple invocation contexts, enabling it to capture different usage patterns and interaction scenarios.

To further examine the trade-off between invocation proximity and caller diversity, Fig. 7 presents the Pass@1 performance of THM and IPS across tasks grouped by their average invocation depth in the full test suite. When the average invocation depth is relatively small (∼1–10, covering 913 tasks, 93.16% of the REPOCOD), THM consistently outperforms IPS. In this regime, test cases provide sufficiently direct interactions with the target function, allowing THM to effectively leverage diverse invocation contexts and expose TENET to richer usage patterns while maintaining manageable reasoning complexity, resulting in higher generation accuracy. As the invocation depth increases (≥ 11, 67 tasks, 6.84% of the REPOCOD), the advantage of THM diminishes. This suggests that when test suites lack close-range invocations of the target function, prioritizing caller diversity may select tests with long and indirect call chains. Such cases may require the agent to traverse deeper execution paths to interpret test behavior, introducing additional abstractions, noisy signals, and increasing reasoning difficulty.

Overall, THM is effective in the vast majority of tasks with moderate invocation depth, where combining caller diversity and invocation proximity provides strong and complementary guidance for code generation. In scenarios with highly indirect invocation test structures, prioritizing shorter call chains may yield cleaner signals and suggest a promising direction.

E. RQ5: The Impact of Test Usage Stage

Tests can be incorporated at different stages of the agent workflow, potentially influencing generation quality and refinement behavior. To better understand when and how tests contribute, we analyze the impact of using tests at different phases in TENET. In the **NoTest** setting, no tests are provided and the RRW is disabled. In **PreGen**, tests are used only during retrieval before the initial generation. In **PostGen**, tests are applied only in the RRW after the initial generation. Finally, in **AllStage**, tests are available throughout the workflow.

TABLE VI: Results of leveraging tests at different phases in TENET's workflow.

Phases	Pass@1 (%)↑	Input Cons.↓	Output Cons.↓	API Calls↓
NoTest	29.90	32,829	2,482	5.04
PreGen	36.93	35,427	2,408	5.63
PostGen	42.65	138,330	4,710	8.91
AllStage	49.18	147,968	4,645	8.28

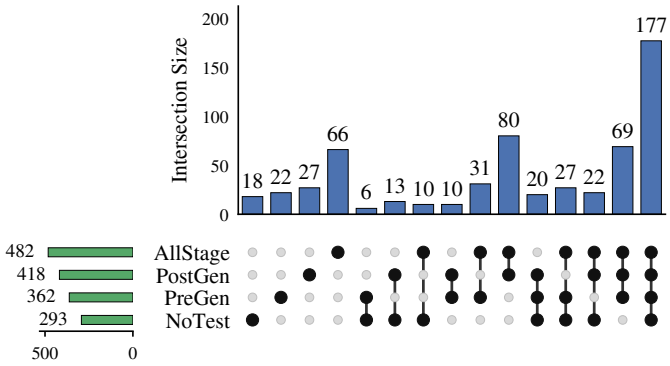


Fig. 8: Overlap of solved tasks on REPOCOD across different test usage stages.

Table VI reports the performance of leveraging tests at different stages of TENET. We make three main observations. **First, leveraging tests across more stages consistently improves the implementation accuracy.** Pass@1 increases from 29.90% (NoTest) to 36.93% (PreGen), 42.65% (PostGen), and peaks at 49.18% with AllStage. AllStage performs best on 10 projects, with `plotly.py` as the only exception where **PostGen** is superior. These results demonstrate the effectiveness of incorporating tests throughout the generation process and highlight the potential of TDD for LLM-based software development. **Second, this improvement comes at a higher cost**. PostGen and AllStage incur substantially higher computational costs than NoTest and PreGen, requiring nearly 100k additional input tokens, roughly twice the number of output tokens, and more API calls, which reflects the extra context retrieval and debugging efforts required by the RRW.

Third, test usage at different stages exhibits complementary effects rather than solving an identical set of tasks. Fig. 8 shows that AllStage achieves the largest number of uniquely solved tasks (66), while a shared core (177) is solved by all settings. Although part of this diversity may arise from the inherent randomness of LLMs, the substantial partial overlaps (e.g., 80 and 69) indicate that different stages provide complementary rather than redundant signals. Notably, PostGen shares larger intersections with AllStage, suggesting

that refinement-time test usage contributes more directly to final correctness. It also points to potential opportunities for improvement by tailoring how tests are leveraged at different stages for different repositories.

VI. Threats to Validity

Generalizability. This work focuses on repository-level code generation under the TDD setting with existing developer-written tests; our findings may not generalize to other settings, such as issue-resolution benchmarks or repositories with incomplete test suites. **Test Information Leakage.** Because existing repository tests are used, evaluation tests may be visible before generation, creating a potential leakage threat: agents may inspect, retrieve, or execute tests that are also used for evaluation. This issue is not unique to TENET but is inherited in repository-level code generation benchmarks where evaluation tests remain in the repository during agent execution [15, 23, 25]. In TENET, we reduce this threat by exposing only THM-selected tests to the downstream agent; all remaining evaluation tests are withheld and used solely for final evaluation. The THM-selected tests may still leak implementation logic if they reveal the target algorithm rather than observable behavior. We manually examined 20 randomly sampled tasks per benchmark and found no such cases: selected tests mainly specify I/O formats, exceptions, relational properties, or concrete outputs. For example, Fig. 3 validates RGB outputs while the core RGB→HLS→RGB transformation remains hidden. Still, tests written after implementation may implicitly encode design decisions; a fuller AI-driven TDD workflow should author tests *before* the implementation exists. **Evaluation Confounds.** Agentic systems differ in control flow, tool use, and configuration. We reduce these confounds by aligning tool-calling capacity, monetary budgets, model temperature, and backbone models, but architectural differences and LLM stochasticity remain; repeated runs would further strengthen robustness. **Metric Limitations.** We define correctness as passing the available tests. Since test suites may be incomplete or flaky, our results should be read as test satisfaction under existing developer-written tests rather than full semantic correctness. In RQ4, THM selects tests using heuristic proxies such as cyclomatic complexity and invocation depth, which may not fully capture test quality or fault-revealing capability.

VII. Conclusion & Future Work

This work introduces TENET, an agentic framework for **repository-level code generation under the TDD paradigm**, comprising (1) a test harness mechanism for selecting concise, effective tests, (2) a tailored toolset for context retrieval and debugging, and (3) a reflection-based refinement workflow for iterative code fixing. TENET achieves 69.08% and 81.77% Pass@1 on REPOCOD and RepoEval respectively with Claude Sonnet 4, outperforming baselines by 9.49 and 2.17 percentage points. We also present the first systematic study of test suite usage in this setting, analyzing test quantity, selection strategies, and usage stages to inform how TDD can be leveraged in agent-based development. Future work includes extending TENET from using existing developer-written tests to actively co-evolving tests with implementations, enabling a fuller realization of test-driven repository-level code generation.

Acknowledgment

This research was supported in part by NSF 1901242 and 2006688. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] K. Beck, *Test driven development: By example*. Addison-Wesley Professional, 2022.

[2] N. S. Mathews and M. Nagappan, "Test-driven development and llm-based code generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1583–1594.

[3] S. Piya and A. Sullivan, "Llm4tdd: Best practices for test driven development using large language models," in *Proceedings of the 1st International Workshop on Large Language Models for Code*, ser. LLM4Code '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 14–21.

[4] Z. Tian, J. Chen, and X. Zhang, " Fixing Large Language Models' Specification Misunderstanding for Better Code Generation ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 645–645.

[5] B. George and L. Williams, "A structured experiment of test-driven development," *Information and Software Technology*, vol. 46, no. 5, pp. 337–342, 2004, special Issue on Software Engineering, Applications, Practices and Tools from the ACM Symposium on Applied Computing 2003.

[6] L. Williams, E. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice," in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, 2003, pp. 34–45.

[7] D. Janzen and H. Saiedian, "Does test-driven development really improve software design quality?" *IEEE Software*, vol. 25, no. 2, pp. 77–84, 2008.

[8] S. V. Sheta, "The role of test-driven development in enhancing software reliability and maintainability," *Journal of Software Engineering (JSE)*, vol. 1, no. 1, pp. 13–21, November 2023, available at SSRN: https://ssrn.com/abstract=5034145 or http://dx.doi.org/10.2139/ssrn.5034145.

[9] Z. Mafi and S.-H. Mirian-Hosseinabadi, "Regression test selection in test-driven development," *Automated Software Engineering*, vol. 31, no. 1, p. 9, 2023.

[10] P. Cassieri, S. Romano, and G. Scanniello, "Generative artificial intelligence for test-driven development: Gai4- tdd," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2024, pp. 902–906.

[11] M. T. Baldassarre *et al.*, "Studying test-driven development and its retainment over a six-month time span," *Journal of Systems and Software*, vol. 176, p. 110937, 2021.

[12] J. T. Liang, C. Yang, and B. A. Myers, "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in *Proceedings of the 46th IEEE/ACM international conference on software engineering*, 2024, pp. 1–13.

[13] Anthropic, "Claude code best practices," https://www.anthropic.com/engineering/claude-code-best-practices, 2025.

[14] B. Cherny, "Claude code & the evolution of agentic coding," https://www.youtube.com/watch?v=Lue8K2jqfKk, Anthropic, 2025, talk at AI Engineer World's Fair, San Francisco. Premiered July 4, 2025. Accessed: 2025-08-31.

[15] S. Liang *et al.*, "Can language models replace programmers? repocod says 'not yet'," arXiv preprint arXiv:2410.21647, 2024.

[16] S. Fakhoury *et al.*, "Llm-based test-driven interactive code generation: User study and empirical evaluation," *IEEE Transactions on Software Engineering*, vol. 50, no. 9, pp. 2254–2268, 2024.

[17] J. Li *et al.*, "Acecoder: An effective prompting technique specialized in code generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, Nov. 2024.

[18] Y. Chen *et al.*, "Can old tests do new tricks for resolving swe issues?" *Proceedings of the ACM on Software Engineering*, vol. 3, no. FSE, pp. 3186–3207, 2026.

[19] J. Liu *et al.*, "Issueexec: A test-driven approach for localizing software engineering issues," *arXiv preprint arXiv:2607.17286*, 2026.

[20] M. Chen *et al.*, "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.

[21] J. Austin *et al.*, "Program synthesis with large language models," arXiv preprint arXiv:2108.07732, 2021.

[22] D. Hendrycks *et al.*, "Measuring coding challenge competence with apps," *NeurIPS*, 2021.

[23] F. Zhang *et al.*, "RepoCoder: Repository-level code completion through iterative retrieval and generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 2471–2484.

[24] Y. Ding *et al.*, "Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion," in *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.

[25] J. Li *et al.*, "DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories," in *Findings of the Association for Computational Linguistics: ACL 2024*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 3603–3614.

[26] Y. Zhang *et al.*, "Learning-based widget matching for migrating gui test cases," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24, vol. 66. ACM, Feb. 2024, p. 1–13.

[27] Z. Z. Wang *et al.*, "Coderag-bench: Can retrieval augment code generation?" 2024.

[28] W. Li *et al.*, "Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2025, pp. 17 160–17 176.

[29] J. Yang *et al.*, "Execrepobench: Multi-level executable code completion evaluation," arXiv preprint arXiv:2412.11990, 2024.

[30] Y. Wang *et al.*, " RLCoder: Reinforcement Learning for Repository-Level Code Completion ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 165–177.

[31] D. Wu *et al.*, "Repoformer: selective retrieval for repository-level code completion," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.

[32] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," in *Proceedings of the 40th International Conference on Machine Learning*, ser. ICML'23. JMLR.org, 2023.

[33] D. Liao *et al.*, "$\mathbf{A^3}$-codgen: A repository-level code generation framework for code reuse with local-aware, global-aware, and third-party-library-aware," *IEEE Transactions on Software Engineering*, vol. 50, no. 12, pp. 3369–3384, 2024.

[34] W. Cheng, Y. Wu, and W. Hu, "Dataflow-guided retrieval augmentation for repository-level code completion," in *ACL*, 2024.

[35] W. Liu *et al.*, "Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model," arXiv preprint arXiv:2406.07003, 2024.

[36] J. Li *et al.*, "Coderag: Supportive code retrieval on bigraph for real-world code generation," arXiv preprint arXiv:2504.10046, 2025.

[37] M. Liang *et al.*, "Repofuse: Repository-level code completion with fused dual context," arXiv preprint arXiv:2402.14323, 2024.

[38] S. Ouyang *et al.*, "Repograph: Enhancing ai software engineering with repository-level code graph," arXiv preprint arXiv:2410.14684, 2025.

[39] H. N. Phan *et al.*, "Repohyper: Search-expand-refine on semantic graphs for repository-level code completion," arXiv preprint arXiv:2403.06095, 2024.

[40] Z. Bi *et al.*, "Iterative refinement of project-level code context for precise code generation with compiler feedback," in *Findings of the Association for Computational Linguistics: ACL 2024*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 2336–2353.

[41] R. Bairi *et al.*, "Codeplan: Repository-level coding using llms and planning," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024.

[42] K. Zhang *et al.*, "CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 13 643–13 658.

[43] X. Wang *et al.*, "OpenHands: An Open Platform for AI Software Developers as Generalist Agents," arXiv preprint arXiv:2407.16741, 2024.

[44] J. Yang *et al.*, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[45] H. Ruan, Y. Zhang, and A. Roychoudhury, " SpecRover: Code Intent Extraction via LLMs ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 617–617.

[46] Y. Zhang *et al.*, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1592–1604.

[47] X. Wang *et al.*, "Executable code actions elicit better llm agents," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.

[48] C. S. Xia *et al.*, "Live-swe-agent: Can software engineering agents self-evolve on the fly?" arXiv preprint arXiv:2511.13646, 2025.

[49] N. F. Liu *et al.*, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.

[50] D. J. Kim, J. Yang, and T.-H. Chen, "A first look at the inheritance-induced redundant test execution," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024.

[51] X. Yuan *et al.*, "debug-gym: A text-based environment for interactive debugging," arXiv preprint arXiv:2503.21557, 2025.

[52] X. Chen *et al.*, "Teaching large language models to self-debug," 2023.

[53] T. X. Olausson *et al.*, "Demystifying gpt self-repair for code generation," 2023.

[54] D. Huang *et al.*, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," 2024.

[55] J. Yang *et al.*, "SWE-bench multimodal: Do AI systems generalize to visual software domains?" in *The Thirteenth International Conference on Learning Representations*, 2025.

[56] J. Yang *et al.*, "Swe-smith: Scaling data for software engineering agents," arXiv preprint arXiv:2504.21798, 2025.

[57] C. E. Jimenez *et al.*, "SWE-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations*, 2024.

[58] M. S. Rashid *et al.*, "Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents," arXiv preprint arXiv:2504.08703, 2025.

[59] DeepSeek-AI *et al.*, "Deepseek-v3 technical report," arXiv preprint arXiv:2412.19437, 2025.

[60] Anthropic, "Claude sonnet 4," https://www.anthropic.com/claude/sonnet, 2025.

[61] M. H. Erol *et al.*, "Cost-of-pass: An economic framework for evaluating language models," arXiv preprint arXiv:2504.13359, 2025.

[62] G. Mialon *et al.*, "Gaia: a benchmark for general ai assistants," arXiv preprint arXiv:2311.12983, 2023.

[63] S. Vijayvargiya *et al.*, "Openagentsafety: A comprehensive framework for evaluating real-world ai agent safety," arXiv preprint arXiv:2507.06134, 2025.