# TDD-Agent: Test-Driven Reasoning for Code Generation

**Anonymous ACL submission**


## Abstract

Large Language Models (LLMs) have achieved remarkable progress in code generation, yet ensuring correctness in complex, repository-level tasks remains challenging. Existing approaches often use generated tests as static post-hoc validators, which limits their ability to guide implementation and may introduce misleading feedback when the tests themselves are incomplete or incorrect. In this paper, we introduce TDD-Agent, which operationalizes the test-driven development paradigm for code generation. TDD-Agent first prompts the model to generate executable tests, encouraging it to clarify expected behaviors before implementation, and then performs iterative dual-track refinement over both the generated code and tests using execution feedback. We first isolate the effect of test-first reasoning through a prompt variant TDD-prompt on LiveCodeBench, where it consistently improves upon reasoning-based prompting baselines. Building on this finding, we evaluate the full TDD-Agent framework on RepoEval, a repository-level benchmark, and show that it consistently outperforms retrieval-based and agent-based baselines. Additional analyses show that iterative refinement improves not only code correctness but also the effectiveness of the generated tests, yielding higher pass rates, coverage, and mutation scores, suggesting that tests can serve as evolving reasoning artifacts rather than fixed validators. Our source code is available at `https://anonymous.4open.science/r/TDD-Agent-Framework-6370/`.


## 1 Introduction

Large Language Models (LLMs) have demonstrated exceptional performance across a spectrum of software engineering tasks, particularly in code generation. The potential of these models has garnered significant attention from both academia and industry, evidenced by the rapid development of commercial foundation models such as the GPT (OpenAI, 2023), Gemini (Team et al., 2025), and Claude (Anthropic, 2024) series, as well as open-source counterparts like Llama (Grattafiori et al., 2024), Gemma (Team et al., 2024), and Qwen (Yang et al., 2024a). Consequently, enhancing the efficacy of LLMs in complex coding scenarios has emerged as a critical research trajectory.

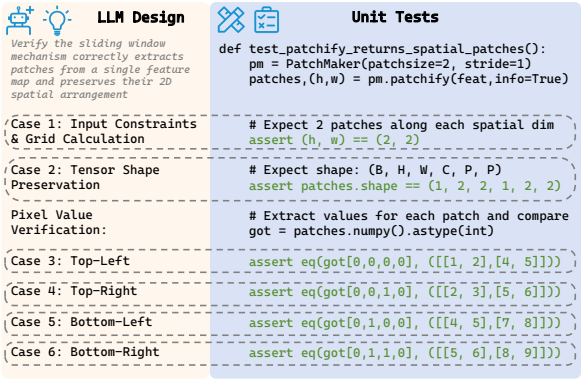


Figure 1: An example demonstrating that formulating tests constitutes a form of reasoning.

To increase model performance, researchers have integrated software engineering methodologies into LLM-based generation. Pair Coder (Zhang et al., 2024a) employs a multi-agent framework separating high-level planning from specific implementation to improve code quality. Beyond planning, integration of software testing has proven particularly effective. Previous work (Mathews and Nagappan, 2024) indicates that providing external test cases significantly boosts model performance on MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021) compared to using problem descriptions alone. Furthermore, AgentCoder (Huang et al., 2024) employs multi-agent systems to decouple the roles of programmer and tester, demonstrating that explicit test generation can enhance code correctness.

However, the reliance on self-generated tests involves inherent risks. Recent work (Chen et al.,

2025) analyzes two paradigms: post-execution and in-execution. Findings reveal that post-execution debugging often suffers from test bias introduced by self-generated tests, leading to misleading feedback. Conversely, in-execution debugging allows LLMs to leverage intermediate states to mitigate this bias. Moreover, the majority of existing test-centric research is confined to function-level tasks. In these scenarios, generating independent functions from natural language descriptions allows for relatively trivial test synthesis (e.g., simple assert statements). It remains an open question whether self-testing strategies remain effective in repository-level contexts, where test generation is substantially more challenging due to dependencies and the need for environment mocking.

A limitation of prior work is that generated tests are often treated as fixed validators after they are produced. Typically, LLMs are directed to focus solely on rectifying implementation code, without the mandate to verify or correct the validity of the tests themselves. We argue that this underuses the potential of testing as a structured reasoning aid. By requiring a model to formulate executable test cases before implementation, the model is encouraged to make its assumptions about inputs, outputs, edge cases, and behavioral constraints explicit. Figure 1 provides an example that illustrates the rationale.

Based on this insight, we instantiate TDD-Agent as a single-agent framework. In this paradigm, a single agent retains conversation history and is granted the autonomy to modify both the code and the tests dynamically. This approach simulates a coherent "test-driven development (TDD)" thought process (Mathews and Nagappan, 2024), allowing the model to iteratively align code and tests without the complexity overhead of coordinating multiple agents.

We empirically evaluate our approach at two levels. First, on LiveCodeBench, we use a lightweight TDD-prompt variant to isolate the effect of test-first reasoning in function-level code generation. Second, on RepoEval (Zhang et al., 2023), we evaluate the TDD-Agent framework in repository-level tasks, where repository navigation and dependency understanding are required.

In summary, our contributions are as follows:

- We propose TDD-Agent, which adapts the test-driven development paradigm for code generation.
- We show that test-first generation provides an executable intermediate representation of task intent.
- We conduct extensive experiments on both function-level and repository-level benchmarks.

## 2 Methodology

### 2.1 Overall Framework

The TDD-Agent framework operates through a two-phase workflow. Let $\mathcal{R}$ denote the repository context, $\mathcal{T}_{tools}$ the set of available tools, $f_{target}$ the target function to be implemented and $\mathcal{A}$ the agent. Figure 2 provides a comprehensive overview of the framework.

**Phase 1: Test-First Specification Setup** In the initial phase, $\mathcal{A}$ acts as a test designer. It leverages tools $\mathcal{T}_{tools}$ to extract relevant context from $\mathcal{R}$ and comprehend the requirements of $f_{target}$. By generating an initial suite of unit tests, denoted as $U_0$, $\mathcal{A}$ is compelled to disambiguate requirements and define precise behavioral boundaries before any implementation logic is written:

$$U_0 = \mathcal{A}_{design}(\mathcal{R}, f_{target} \mid \mathcal{T}_{tools}). \quad (1)$$

This phase concludes with the submission of $U_0$, ensuring that the agent has established a clear, executable understanding of the task.

**Phase 2: Dual-Track Test-Code Co-Refinement** The second phase constitutes an iterative loop of execution and refinement. Building upon the specifications in $U_0$, $\mathcal{A}$ generates the initial implementation code $C_0$ for $f_{target}$. The implementation is then executed against $U_0$ to produce a test report $E_0$:

$$E_t = \text{Execute}(C_t, U_t). \quad (2)$$

Based on the execution feedback $E_t$, $\mathcal{A}$ performs a reflection process.

- If execution succeeds: $\mathcal{A}$ will receive a prompt suggesting that it consider strengthening or expanding the test suite. If $\mathcal{A}$ is confident in the correctness and robustness of $C_t$, it can choose to invoke the `Finish` tool to terminate the task early.
- If execution fails: $\mathcal{A}$ will receive a prompt encouraging it to analyze the cause. Unlike previous approaches that treat unit tests as immutable constraints, our framework enables the dual refinement of both **code** and **tests**.

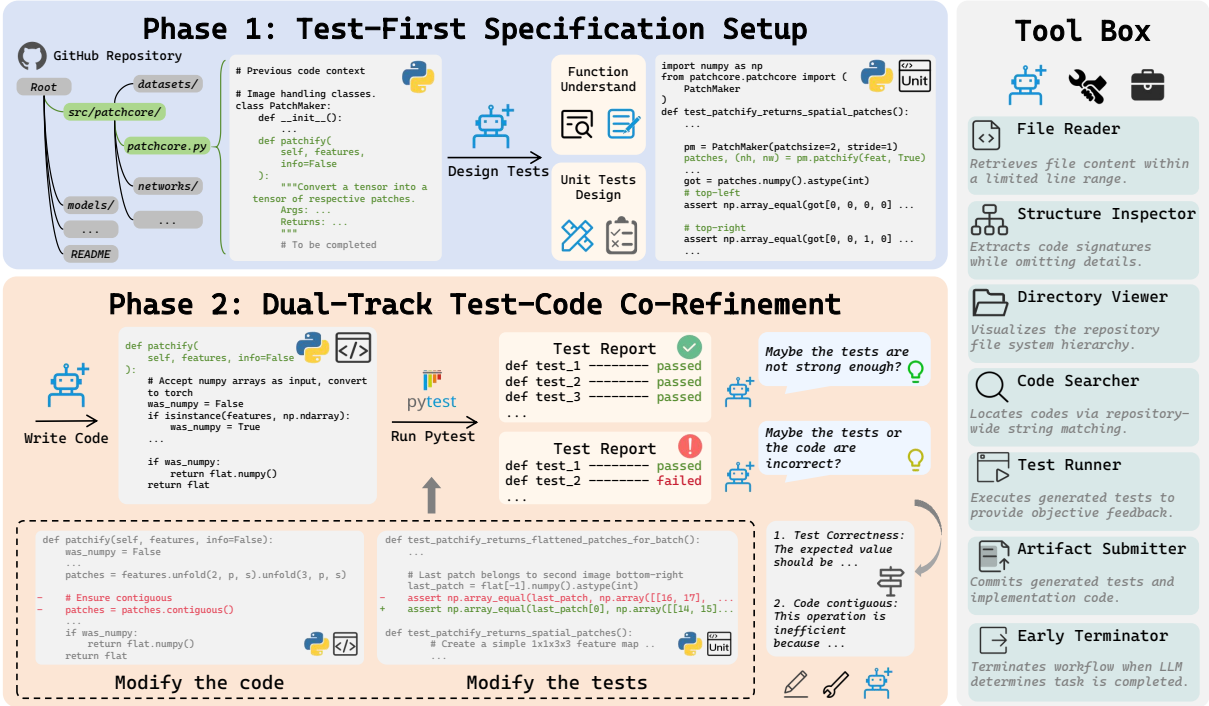


Figure 2: **Overview of the TDD-Agent.** (1) The LLM first designs and generates unit tests for the target function. (2) The LLM generates the complete implementation of the target function, and through execution and refinement, iteratively refines the tests and code.

The state update for the next iteration is formulated as:

$$(C_{t+1}, U_{t+1}) = \mathcal{A}_{reflect}(C_t, U_t, E_t, \mathcal{R}) \quad (3)$$

The iterative process continues until a maximum number of rounds is reached (set to 10) or when $\mathcal{A}$ invokes the `Early Terminator` tool to terminate early.

### 2.2 Tool Box

TDD-Agent is equipped with a lightweight set of tools:

**Context Inspection.** Directory Viewer lists the repository structure, Structure Inspector extracts the skeleton of a file (e.g., class and function) with concrete implementations folded, File Reader retrieves specific code segments based on exact line numbers, Code Searcher performs lexical search using `grep` and `find`.

**Artifact Submission and Test Runner.** The agent utilizes Artifact Submission to submit both the generated tests and implementation code. The generated tests are saved to a temporary file within the same directory as the target file, while the implementation code directly modifies the target file in place. Test Runner strictly executes only the latest version of the generated temporary test file by running pytest. Throughout the prediction phase, the original repository tests are neither executed nor visible to the agent.

**Early Terminator.** At the end of each iteration, specifically, upon receiving the execution results from the Test Runner, the agent can invoke this tool to exit early if it determines that the task has been completed.

## 3 Experiments

We conduct experiments on function-level and repository-level tasks using three high-performing LLMs. **GPT-5-mini** (OpenAI, 2025) is a closed-source model with superior reasoning capabilities. **DeepSeek-V3.2** (DeepSeek-AI et al., 2025) is a general-purpose conversational model with 671B total parameters. **Qwen3-Coder-30B-A3B-Instruct** (Yang et al., 2025) is an open-source Mixture-of-Experts model specialized for coding tasks. We access GPT and DeepSeek via APIs, while Qwen is deployed locally on one NVIDIA H20 GPU using vLLM (Kwon et al., 2023). The detailed generation parameters for each model are in Appendix A.

### 3.1 Function-Level Study

We design a prompting variant TDD-prompt, which asks the LLM to formulate tests before producing the final implementation. We conduct comparative

| Method | GPT | DeepSeek | Qwen |
|---|---|---|---|
| one-shot | 53.08 | 45.40 | 39.60 |
| CoT-prompt | 67.41 | 67.46 | 42.99 |
| SCoT-prompt | 66.83 | 64.20 | 40.80 |
| Self-Planning | 68.17 | 66.70 | 42.68 |
| ICoT-prompt | 68.48 | 62.23 | 41.88 |
| **TDD-prompt** | **70.04** | **67.86** | **44.87** |

Table 1: Pass@1 (%) comparison on LiveCodeBench across three models. Bold and underline indicate the best and second-best performance.

| Method | GPT | DeepSeek | Qwen |
|---|---|---|---|
| In-File | 44.84 | 39.34 | 32.75 |
| RAG | 50.55 | 46.15 | 35.16 |
| RepoCoder | 55.16 | 50.55 | 41.10 |
| mini-SWE-agent | 61.31 | 84.18 | 52.97 |
| **TDD-Agent-iter5** | 77.36 | 88.13 | 58.46 |
| **TDD-Agent-iter10** | **78.24** | **90.77** | **59.34** |

Table 2: Performance comparison on RepoEval across three models. Numbers are presented in percentage (%). Bold and underline indicate the best and second-best among all compared methods.

experiments on **LiveCodeBench** (Jain et al., 2024) to isolate the effect of test-first reasoning. The detailed experiment settings are in Appendix B.

**Baselines.** **CoT** (Wei et al., 2022) asks LLMs to generate natural language reasoning before generating the code. **SCoT** (Li et al., 2025a) asks LLMs to use programming structures to generate structured reasoning steps before generating the code. **Self-Planning** (Jiang et al., 2024) asks LLMs to first generate a sub-task plan and then uses it to guide code generation. **ICoT** (Li et al., 2025c) asks LLMs to first capture the task intention and then uses it to guide code generation.

**Results.** To mitigate computational cost and account for inherent randomness, we generate **10** samples for each problem and measure the unbiased pass@1 metric (Chen et al., 2021). As shown in Table 1, TDD-prompt outperforms all baselines on three LLMs. This consistent improvement demonstrates that TDD-prompt functions as a reasoning framework.

### 3.2 Repository-Level Experimental Setup

**Datasets.** We evaluate our method on the RepoEval dataset (Zhang et al., 2023), which comprises repository-level line, API, and function completion tasks. In this study, we focus specifically on the function completion subset, derived from eight distinct Python repositories and containing a total of 455 problems.

**Metrics.** Following the protocol of RepoCoder, we employ an execution-based evaluation metric. We utilize the repository's existing unit tests to assess functional correctness. Specifically, we integrate the generated code into the original repository and execute the accompanying unit tests. A solution is deemed successful (*Pass*) only if it passes all associated test cases; otherwise, it is marked as failed.

**Execution Environment.** We construct a dedicated Docker environment for each repository, where all required dependencies are pre-installed and configured. Before evaluation, we verify that the original test suite provided by each repository can be executed successfully in the corresponding environment. During prediction and evaluation, each Docker container is allocated 2 CPU cores and 4 GB of memory. Each command is limited to a maximum execution time of 600 seconds. If an evaluation run exceeds the timeout, it is treated as a failure.

### 3.3 Repository-Level Baselines

**In-File Completion.** We provide the LLM exclusively with the code context available within the current file (prefix code) and require it to complete the missing function body.

**RAG.** Following RepoCoder, we employ a sparse bag-of-words model as the retriever (Lu et al., 2022). This model tokenizes both the query and candidate snippets to calculate similarity via the Jaccard index (Jaccard, 1912).

**RepoCoder.** RepoCoder is an iterative retrieval and generation framework. We adopt exactly the same configuration specified in the original paper.

**mini-SWE-agent** (Yang et al., 2024b) is a widely adopted software engineering agent equipped with a bash tool. It achieves excellent performance on **SWE-Bench** and is recommended by its original authors. We set the maximum number of LLM calls to 100 for each problem.

### 3.4 Repository-Level Main Results

Experimental results presented in Table 2 demonstrate that **TDD-Agent** outperforms all baselines. The detailed results for each repository are reported in Appendix C.

While retrieval-based methods like RAG and RepoCoder generally enhance performance by providing relevant context, their improvements are relatively modest. In contrast, agent-based meth-

| Model | Method | Prompt | Completion | Calls |
|---|---|---|---|---|
| GPT | mini | 115.76 | 1.32 | 11.95 |
| | TDD-iter5 | 120.48 | 1.62 | 18.62 |
| | TDD-iter10 | 155.61 | 1.71 | 22.92 |
| DS | mini | 390.10 | 6.38 | 31.61 |
| | TDD-iter5 | 361.64 | 9.71 | 29.39 |
| | TDD-iter10 | 403.01 | 10.66 | 31.49 |
| Qwen | mini | 106.83 | 2.14 | 14.81 |
| | TDD-iter5 | 117.07 | 3.02 | 15.34 |
| | TDD-iter10 | 129.87 | 3.22 | 17.30 |

Table 3: Average token usage (in thousands) and average LLM calls across different models on mini-SWE-agent and TDD-Agent.

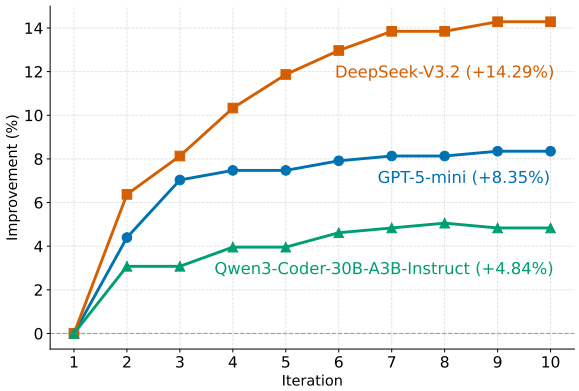


Figure 3: The pass rate improvement across iterations on three models.

ods like mini-SWE-agent and TDD-Agent achieve stronger performance gains. Notably, TDD-Agent already surpasses the mini-SWE-agent baseline by its fifth iteration, and increasing the number of iterations yields further performance gains.

For agent-based methods, efficiency is a crucial metric alongside overall performance. We report the average token usage and LLM calls for mini-SWE-agent and TDD-Agent in Table 3. We observe that at its fifth iteration, TDD-Agent exhibits comparable token consumption to mini-SWE-agent, while consistently achieving superior performance. The detailed token usage and time cost are presented in Appendix C.

# 4 Analysis

## 4.1 Early Termination

Figure 4 reports the Match Rate and End Rate across all iterations for the three models. Match Rate denotes the proportion of tasks whose current generated code passes the current generated tests, while End Rate denotes the proportion of tasks for which the model chooses to terminate at the current iteration.

The results reveal clear differences across models. In the first iteration, DeepSeek behaves the most conservatively. Although a substantial portion of its generated implementation already passes the generated tests, it rarely chooses to terminate. This indicates that DeepSeek does not simply treat self-test success as sufficient evidence of correctness. Instead, it tends to continue refining even after the current tests are passed. This conservative behavior helps explain why DeepSeek obtains the largest improvement through iterative refinement, as it keeps using additional rounds to strengthen the test suite and correct potential hidden defects.

GPT and Qwen terminate more aggressively in early iterations. For GPT, the End Rate remains consistently lower than the Match Rate, suggesting that the model generally requires self-test success before deciding to stop. This indicates a stopping heuristic: passing the generated tests is treated as an important but not always sufficient condition for termination. Qwen exhibits a different pattern. From the middle iterations onward, its End Rate becomes higher than its Match Rate. This means for some tasks, Qwen chooses to stop even when the current implementation has not passed its generated tests.

## 4.2 Improvement Across Iterations

Figure 3 illustrates the pass rate improvement achieved by TDD-Agent across iterations for the three models. All models benefit from iterative refinement, confirming that the execution-feedback loop is effective for improving performance.

Among the three models, DeepSeek achieves the largest overall gain. This is consistent with the early-termination analysis in Section 4.1: DeepSeek is less likely to stop after the first successful self-test and instead continues to refine its artifacts. As a result, it can exploit later iterations more effectively, leading to the largest cumulative improvement.

GPT also shows a clear and stable improvement trend. Its performance increases rapidly during the first several rounds and then gradually stabilizes. This suggests that GPT can effectively use execution feedback to correct errors, while its termination behavior prevents excessive premature stopping.

Qwen obtains a smaller but still positive gain. Although the analysis in Figure 4 indicates that Qwen sometimes terminates before passing its generated tests, its performance still improves with

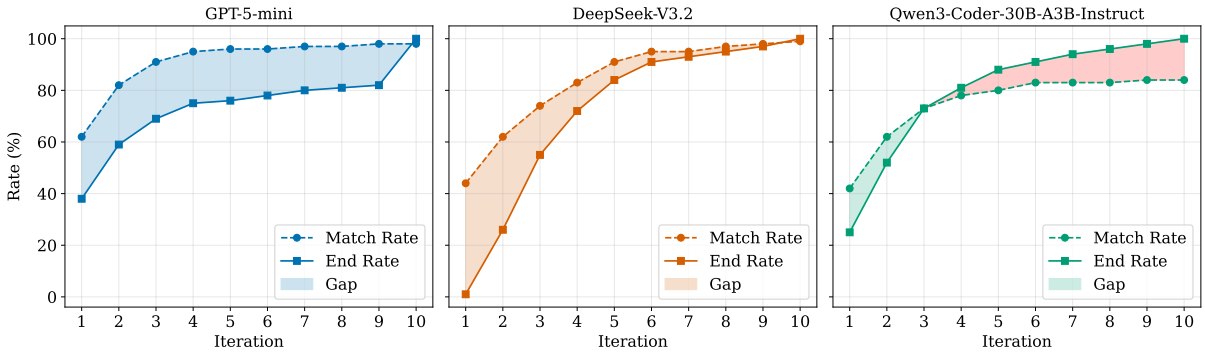


Figure 4: Comparison between End Rate and Match Rate across different iteration rounds on three models. Match Rate represents the proportion of generated code that successfully passes the generated tests. End Rate indicates the proportion of tasks where the model autonomously decides to terminate.

more iterations, implying that although some tasks suffer from early stopping, the remaining active tasks can still benefit from continued refinement.

Across all three models, the improvement curves gradually converge and reach a near-plateau around iterations 6–7. This trend indicates that most useful corrections are made in the early and middle stages of the iterative process, while additional rounds provide diminishing returns. This observation suggests that a moderate iteration budget can provide a favorable trade-off between performance and computational overhead.

### 4.3 Quality of Generated Tests

A core premise of TDD-Agent is that generated tests should not be treated as static artifacts. Instead, they should evolve together with the implementation. To examine whether this property holds in practice, we analyze the quality of generated tests from three perspectives: pass rate, coverage rate and mutation score.

At each iteration, the generated test suite is executed against the ground-truth implementation. A test suite is considered passed if all its tests succeed on the ground-truth implementation. A low pass rate indicates potential issues such as incorrect assumptions or hallucinated requirements.

The coverage rate is measured using the `pytest-cov` plugin. Specifically, we execute the generated tests on the ground-truth implementation and compute line coverage for the target function:

$$\text{Coverage Rate} = \frac{\text{Covered Lines}}{\text{Total Lines}}. \quad (4)$$

This metric measures how thoroughly the generated tests exercise the implementation logic.

We assess mutation score using **MutPy** (Hossner et al., 2021). For each problem, we generate 20 mutants of the ground-truth implementation and execute the LLM-generated tests against them. The mutation score is defined as the fraction of mutants killed by the generated tests:

$$\text{Mutation Score} = \frac{\text{Killed Mutants}}{\text{Total Mutants}}. \quad (5)$$

If the generated tests fail on the ground-truth implementation, we set the mutation score to 0.

Figure 5 presents the pass rate, coverage rate, and mutation score of generated tests across iterations. The results show that all three metrics generally improve as the number of iterations increases. This indicates that the iterative refinement process improves not only the generated code but also the generated tests. Through iterative refinement, the agent gradually removes invalid assertions, corrects mismatched expectations, and expands the test suite to cover more execution paths.

The tests generated by DeepSeek achieve consistently high values across all three metrics, which provides a direct explanation for its large code-generation gain. Since the generated tests become more reliable, they provide more accurate feedback, enabling the model to correct defects more effectively. GPT also shows steady improvements, suggesting that it can consistently refine its generated tests based on execution feedback. Qwen starts with comparatively lower metric values and obtains a smaller gain, but still shows a positive trend. This further confirms that even when a model exhibits early-termination behavior, the dual-track refinement process can still improve the quality of its generated tests.

These findings support the central hypothesis of TDD-Agent: generated tests are most effective when they are treated not as fixed validators but as evolving reasoning artifacts.

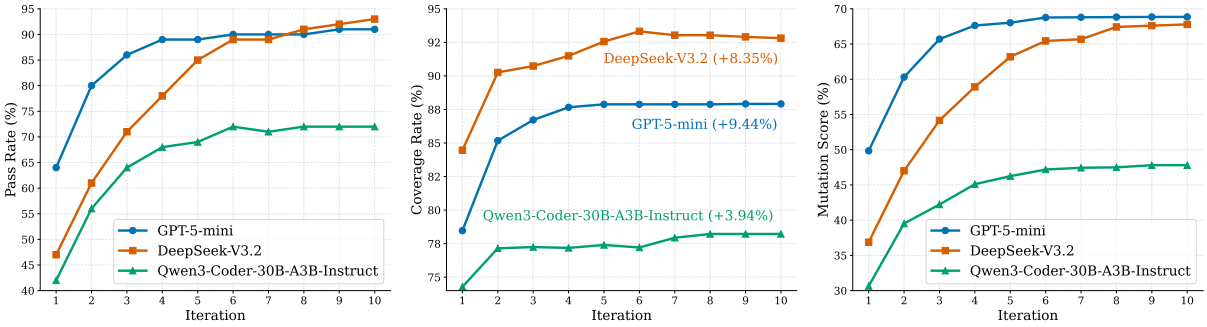


Figure 5: Pass rate, coverage rate, and mutation score of generated tests over iterations for three models.

## 4.4 Ablation Study

To disentangle the contribution of each core component in TDD-Agent, we conduct an ablation study on RepoEval. TDD-Agent consists of two key designs: (1) test-first generation, which requires the model to construct executable specifications before implementation, and (2) dual-track refinement, which allows the model to iteratively revise both the implementation and the generated tests based on execution feedback.

We compare TDD-Agent with four variants. *Vanilla* directly generates the implementation without test generation or iterative refinement. *Reflect* removes test generation and performs iterative self-reflection only on the implementation. *Single-track* keeps test-first generation but freezes the generated tests during refinement, allowing only the code to be modified. *TDD-Agent-iter1* executes only the first test generation and code generation without subsequent refinement.

As shown in Table 4, the full TDD-Agent consistently achieves the best performance, demonstrating that both test-first reasoning and dual-track refinement are essential to the framework. Compared with *Vanilla*, *TDD-Agent-iter1* achieves higher pass rates, indicating that generating tests before implementation can indeed improve code quality by encouraging the model to clarify expected behaviors in advance. However, the improvement is limited, suggesting that a single round of test-first generation is insufficient.

The comparison between *Vanilla* and *Reflect* shows that iterative reflection alone can also improve performance. However, reflection alone remains limited because the model lacks concrete feedback to verify whether its revisions are actually correct. *Single-track* introduces test-first generation and therefore provides executable feedback during refinement. Nevertheless, its improvement is also limited, since the initially generated tests may contain errors or incomplete assumptions, and thus may fail to provide accurate guidance in some cases.

| Variant | GPT | DeepSeek | Qwen |
|---|---|---|---|
| Vanilla | 68.35 | 73.63 | 52.97 |
| Reflect | <u>72.09</u> | <u>81.53</u> | 54.06 |
| Single-track | 70.11 | 79.56 | <u>57.36</u> |
| TDD-Agent-iter1 | 69.89 | 76.48 | 54.51 |
| **TDD-Agent** | **78.24** | **90.77** | **59.34** |

Table 4: Ablation study on RepoEval. Results are reported as pass rate (%). Bold and underline indicate the best and second-best performance for each model.

The gap between *Single-track* and the full TDD-Agent highlights the importance of dual-track refinement. Allowing the agent to revise both tests and code enables it to correct not only implementation errors but also flawed or insufficient tests. This refinement process helps the generated tests gradually better align with the intended behavior of the target function, thereby providing more reliable feedback for subsequent code revisions. Overall, these results confirm that test generation is most effective when it is treated not as a fixed validator, but as an evolving reasoning artifact that is refined jointly with the implementation.

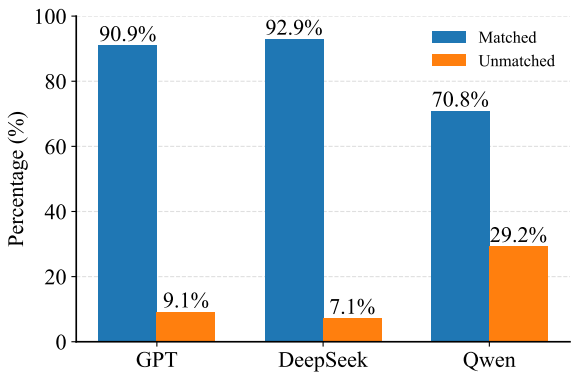


Figure 6: Distribution of matched and unmatched failures across three backbone models.

### 4.5 Failure Case Analysis

A natural concern in TDD-Agent is the *credit assignment problem*. When an execution fails, the agent must determine whether the implementation is flawed or the tests are incorrect. There are cases where the agent may fix correct code to pass a hallucinated test, or conversely, discard valid tests to accommodate erroneous code. However, our statistics show that such cases account for less than 10% of the failed cases.

We further analyze the failed cases according to whether the implementation passes the generated tests at termination. As shown in Figure 6, most failures are matched failures, where the generated implementation passes the generated tests but fails the held-out repository tests. This indicates a form of false-positive verification: the agent has produced tests that are consistent with its implementation, yet fail to fully capture the intended behavior, acting as an incomplete or misaligned executable specification. Since the refinement loop is driven by execution feedback from the current generated tests, defects that are not exercised by these tests may remain invisible throughout refinement. As a result, tests and code can co-evolve into an internally consistent but incomplete state: the implementation satisfies the generated tests, while both artifacts still fail to capture behaviors required by the repository oracle. This suggests that matched failures reflect not merely low test quality, but a limitation in the model's task understanding as operationalized through test generation. More detailed statistics are reported in Appendix C.

## 5 Related Work

**Test-Guided Code Generation.** TICODER (Fakhoury et al., 2024) introduces an interactive workflow where models generate candidate tests to clarify user intent, while INTUT (Nan et al., 2025) uses test intentions to improve branch coverage in unit test generation. COCOEVO (Li et al., 2025b) co-evolves programs and test cases via genetic search, and LLMLOOP (Ravi et al., 2025) incorporates mutation analysis to make generated tests more challenging. More closely related, TENET (Hu et al., 2025) leverages pre-existing repository tests for test selection, context retrieval, and feedback-guided code refinement. In contrast to prior work that relies on user interaction, predefined tests, or uses tests mainly as validation/refinement signals, TDD-Agent treats test generation as a reasoning step: it synthesizes executable tests before implementation and jointly refines both tests and code through execution feedback, without assuming access to predefined tests during prediction.

**Software Engineering Agents.** Yang et al. (2024b) introduce SWE-AGENT, which utilizes a custom Agent-Computer Interface to prevent the model from being overwhelmed by verbose shell outputs. Zhang et al. (2024b) propose CODEAGENT, which integrates symbol navigation and testing tools, demonstrating that precise code navigation is critical for repository-level tasks. Tao et al. (2024) propose MAGIS, a framework that assigns distinct roles to agents to simulate a real-world development lifecycle. Conversely, Xia et al. (2024) argue for simplicity with AGENTLESS, a method that eschews agentic loops for a simpler workflow. OPENHANDS (Wang et al., 2025a) further accelerates this field by providing a unified runtime for developing and evaluating such generalist agents.

**Retrieval-Augmented Completion.** RLCODER (Wang et al., 2025b) employs reinforcement learning to align the retriever with code generation utility. COCO (Zhao et al., 2025) parses code at multiple granularities to capture structural dependencies.

## 6 Conclusion and Future Work

In this paper, we introduced **TDD-Agent**, which operationalizes the Test-Driven Development paradigm. TDD-Agent treats test generation as a process of reasoning, compelling the model to clarify requirements and define executable boundaries prior to implementation. Through iterative refinement, our framework enables the dual refinement of both code and tests. Experimental results demonstrate that TDD-Agent consistently outperforms baselines. Our analysis further reveals that the quality of generated tests improves concurrently with the code implementation, validating the efficacy of our dual-track refinement strategy. For future work, we plan to extend TDD-Agent to more programming languages and further improve the reliability of self-generated tests. In particular, our failure case analysis suggests that many remaining failures arise when generated tests fail to expose hidden defects in the implementation. Future work may therefore explore stronger test-oracle construction, mutation-guided test expansion, and uncertainty-aware termination criteria to reduce false-positive verification.

## Limitations

**Scope of Evaluation Languages.** Our experimental validation is currently confined to Python. To adapt our framework to other languages, one only needs to replace the environment-specific components (e.g., JUnit for Java, Jest for JavaScript), while the high-level reasoning and iteration logic remain unchanged.

**Limited Repository Understanding.** TDD-Agent currently relies on a lightweight tool set. While these tools are sufficient to support iterative refinement, they provide limited semantic access to repository-level information. This limitation is reflected in our failure analysis: most failures are matched failures. Such cases suggest that the dominant bottleneck is incomplete understanding of tasks and repositories. Future work could integrate stronger semantic code search, dependency-aware navigation, call-graph analysis and more informative repository-level context construction to improve the quality of generated tests and reduce false-positive verification.

**Reliability of Execution Feedback.** Our framework assumes that the test execution provides reliable feedback. In practice, flaky tests or nondeterministic behavior may introduce noisy signals. Such unreliable feedback can mislead the agent into modifying correct implementations or discarding valid tests, thereby degrading the effectiveness of iterative refinement. Future work could improve robustness by incorporating repeated test execution and flaky-test detection.

**Computational Overhead.** The iterative nature of TDD-Agent inherently incurs higher token consumption and latency.

## References


Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. *Anthropic Technical Report*.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Xiancai Chen, Zhengwei Tao, Kechi Zhang, Changzhi Zhou, Xinyu Zhang, Wanli Gu, Yuanpeng He, Mengdi Zhang, Xunliang Cai, Haiyan Zhao, and Zhi Jin. 2025. Revisit self-debugging with self-generated tests for code generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 18003–18023, Vienna, Austria. Association for Computational Linguistics.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, and 245 others. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *Preprint*, arXiv:2512.02556.

Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Llm-based test-driven interactive code generation: User study and empirical evaluation. 50(9):2254–2268.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Philipp Hossner, Konrad Hałas, Steven Myint, and Andreas Mueller. 2021. MutPy: a mutation testing tool for python 3.x source code. `https://github.com/mutpy/mutpy`. Accessed: 2026-05-18.

Yiran Hu, Nan Jiang, Shanchao Liang, Yi Wu, and Lin Tan. 2025. Tenet: Leveraging tests beyond validation for code generation. *Preprint*, arXiv:2509.24148.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *Preprint*, arXiv:2312.13010.

Paul Jaccard. 1912. The distribution of the flora in the alpine zone. *The New Phytologist*, 11(2):37–50. JSTOR: 2427226.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *Preprint*, arXiv:2403.07974.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7).

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Evan Zheng, Cody Hao Yu, Joseph Gonzalez, Ion Stoica, and Zhanghao Wu. 2023. Efficient

memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025a. Structured chain-of-thought prompting for code generation. *ACM Trans. Softw. Eng. Methodol.*, 34(2).

Kefan Li, Yuan Yuan, Hongyue Yu, Tingyu Guo, and Shijie Cao. 2025b. Cocoevo: Co-evolution of programs and test cases to enhance code generation. *Preprint*, arXiv:2502.10802.

Shen Li, Li Huang, Shaoxiong Zhan, Weifeng Sun, Tao Yin, Zhongxin Liu, and Meng Yan. 2025c. Intention chain-of-thought prompting with dynamic routing for code generation. *Preprint*, arXiv:2512.14048.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *Preprint*, arXiv:2203.07722.

Noble Saji Mathews and Meiyappan Nagappan. 2024. Test-driven development and llm-based code generation. ASE '24, page 1583–1594, New York, NY, USA. Association for Computing Machinery.

Zifan Nan, Zhaoqiang Guo, Kui Liu, and Xin Xia. 2025. *Test Intention Guided LLM-Based Unit Test Generation*, page 1026–1038. IEEE Press.

OpenAI. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

OpenAI. 2025. GPT-5 mini. `https://developers.openai.com/api/docs/models/gpt-5-mini`. OpenAI API model documentation.

Ravin Ravi, Dylan Bradshaw, Stefano Ruberto, Gunel Jahangirova, and Valerio Terragni. 2025. Llmloop: Improving llm-generated code and tests through automated iterative feedback loops. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 930–934.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. 2024. Magis: Llm-based multi-agent framework for github issue resolution. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA. Curran Associates Inc.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy Lillicrap, Angeliki Lazaridou, and 1332 others. 2025. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Pier Giuseppe Sessa, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, and 89 others. 2024. Gemma: Open models based on gemini research and technology. *Preprint*, arXiv:2403.08295.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025a. Openhands: An open platform for ai software developers as generalist agents. *Preprint*, arXiv:2407.16741.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2025b. *RLCoder: Reinforcement Learning for Repository-Level Code Completion*, page 1140–1152. IEEE Press.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *Preprint*, arXiv:2407.01489.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, and 43 others. 2024a. Qwen2 technical report. *Preprint*, arXiv:2407.10671.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024b. Swe-agent: agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA. Curran Associates Inc.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level

code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. 2024a. A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement. ASE '24, page 1319–1331, New York, NY, USA. Association for Computing Machinery.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024b. 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)*, pages 13643–13658, Bangkok, Thailand. Association for Computational Linguistics.

Xinkui Zhao, Rongkai Liu, Yifan Zhang, Chen Zhi, Lufei Zhang, Guanjie Cheng, Yueshen Xu, Shuiguang Deng, and Jianwei Yin. 2025. Completion by comprehension: Guiding code generation with multi-granularity understanding. *Preprint*, arXiv:2512.04538.

## A Generation Parameter Settings

In our experiments, we specifically utilized the **gpt-5-mini-2025-08-07** for GPT-5-mini. For Qwen3-Coder-30B-A3B-Instruct, we adopt the optimal generation parameters recommended by the official guidelines. Additionally, the temperature parameter for DeepSeek-V3.2 is set to its default value of 1.0.

| Model | Parameter | Value |
|---|---|---|
| GPT-5-mini | temperature | 1.0 |
| | max_tokens | 4096 |
| | reasoning_effort | minimal |
| | verbosity | low |
| DeepSeek-V3.2 | temperature | 1.0 |
| | max_tokens | 4096 |
| Qwen | temperature | 0.7 |
| | top_p | 0.8 |
| | top_k | 20 |
| | repetition_penalty | 1.05 |
| | max_tokens | 4096 |

Table 5: Generation parameter settings used in experiments for all models.

## B Detailed settings of LiveCodeBench

### Dataset and Evaluation Metric

We select LeetCode-sourced problems from the most recent one-year window of **LiveCodeBench** releases (May 1, 2024 to May 1, 2025), resulting in a total of 224 problems. We measure the effectiveness using the pass@k metric (Chen et al., 2021), which provides a robust and execution-based evaluation of functional correctness. By calculating the expected probability that at least one out of $k$ generated samples passes all tests, this metric effectively mitigates the high variance introduced by the stochastic nature of LLMs during decoding. The unbiased estimator of pass@k is defined as:

$$pass@k = \mathbb{E}_{problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]. \quad (6)$$

### Sampling Settings

Considering cost efficiency, we generate **10** candidates per problem. Following recent work (Li et al., 2025c), for single-stage approaches such as one-shot and CoT, we apply the previously described sampling settings. For multi-stage approaches, including SCoT, ICoT, and Self-Planning, we use these same settings to generate 10 reasoning chains during the reasoning phase, followed by deterministic code generation at a temperature of 0.

### Ablation Study

To further evaluate the effectiveness of test-first paradigm, we remove the test generation instruction from the TDD-prompt. The results shown in Table 6 further highlight that the test-generation-first paradigm is a critical factor, as removing the test generation instruction leads to a drop in performance.

| Method | GPT | DeepSeek | Qwen |
|---|---|---|---|
| **TDD-prompt** | **70.04** | **67.86** | **44.87** |
| w/o Test Generation | 68.39 | 67.28 | 43.84 |

Table 6: Ablation study results on LiveCodeBench across three models.

| ID | Link | Count |
|---|---|---|
| 1 | leopard-ai/betty | 36 |
| 2 | CarperAI/trlx | 46 |
| 3 | lucidrains/imagen-pytorch | 67 |
| 4 | deepmind/tracr | 146 |
| 5 | google/lightweight_mmm | 64 |
| 6 | amazon-science/inspection | 32 |
| 7 | facebookresearch/omnivore | 22 |
| 8 | maxhumber/redframes | 42 |

Table 8: Detailed information of the GitHub repositories used for RepoEval.

| Method | Repo | | | | | | | | All |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |
| **GPT-5-mini** | | | | | | | | | |
| In-File | 63.89 | 67.39 | 61.19 | 39.73 | 28.12 | 50.00 | 40.91 | 19.05 | 44.84 |
| RAG | 69.44 | 76.09 | 59.70 | 45.21 | 32.81 | 50.00 | 45.45 | 40.48 | 50.55 |
| RepoCoder | 72.22 | 78.26 | 70.15 | 50.68 | 39.06 | 50.00 | 40.91 | 42.86 | 55.16 |
| mini-SWE-agent | 80.56 | 69.57 | 68.66 | 58.22 | 46.88 | 65.63 | 63.64 | **52.38** | 61.31 |
| TDD-Agent | **86.11** | **93.48** | **92.54** | **74.66** | **73.44** | **81.25** | **72.73** | **52.38** | **78.24** |
| **DeepSeek-V3.2** | | | | | | | | | |
| In-File | 50.00 | 54.35 | 55.22 | 36.30 | 25.00 | 43.75 | 40.91 | 16.67 | 39.34 |
| RAG | 52.78 | 71.74 | 64.18 | 38.36 | 28.13 | 59.38 | 40.91 | 30.95 | 46.15 |
| RepoCoder | 63.89 | 63.04 | 67.16 | 47.95 | 32.81 | 59.38 | 36.36 | 35.71 | 50.55 |
| mini-SWE-agent | 91.67 | 93.48 | 89.55 | 84.25 | 78.13 | **93.75** | **86.36** | 59.52 | 84.18 |
| TDD-Agent | **94.44** | **95.65** | **95.52** | **91.78** | **90.62** | **93.75** | 68.18 | **80.95** | **90.77** |
| **Qwen3-Coder-30B-A3B-Instruct** | | | | | | | | | |
| In-File | 38.89 | 43.48 | 47.76 | 33.56 | 15.62 | 40.62 | 27.27 | 11.90 | 32.75 |
| RAG | 52.78 | 54.35 | 40.30 | 28.08 | 17.19 | 53.12 | 18.18 | 38.10 | 35.16 |
| RepoCoder | 58.33 | 56.52 | 49.25 | 34.25 | 31.25 | 53.12 | 27.27 | 33.33 | 41.10 |
| mini-SWE-agent | 66.67 | 58.70 | 65.67 | 54.11 | **46.88** | 59.38 | 40.91 | 21.43 | 52.97 |
| TDD-Agent | **77.78** | **71.74** | **73.13** | **58.21** | 40.63 | **81.25** | **45.46** | **30.95** | **59.34** |

Table 7: Performance comparison on RepoEval. Numbers are presented in percentage (%), with the best performance highlighted in bold.

| Iter | GPT-5-mini | | DeepSeek-V3.2 | | Qwen3-Coder-30B-A3B-Instruct | | | |
|---|---|---|---|---|---|---|---|---|
| | Prompt | Completion | Prompt | Completion | Prompt | Completion | Cached | Total |
| 1 | 56674 | 1111 | 158337 | 3637 | 59054 | 1685 | 53510 | 7230 |
| 2 | 80363 | 1371 | 236649 | 5961 | 85475 | 2342 | 79185 | 8633 |
| 3 | 98505 | 1516 | 295616 | 7895 | 102012 | 2714 | 95369 | 9357 |
| 4 | 110656 | 1585 | 335806 | 9042 | 110371 | 2898 | 103563 | 9705 |
| 5 | 120478 | 1621 | 361637 | 9711 | 117065 | 3024 | 110155 | 9934 |
| 6 | 129245 | 1652 | 378503 | 10117 | 121636 | 3103 | 114669 | 10071 |
| 7 | 137075 | 1673 | 387326 | 10327 | 124893 | 3156 | 117883 | 10166 |
| 8 | 143976 | 1690 | 394722 | 10490 | 127595 | 3193 | 120549 | 10239 |
| 9 | 149775 | 1702 | 399190 | 10592 | 129062 | 3215 | 121998 | 10279 |
| 10 | 155611 | 1711 | 403252 | 10660 | 129866 | 3222 | 122793 | 10295 |

Table 9: Average token usage of TDD-Agent across iterations for three models on RepoEval.

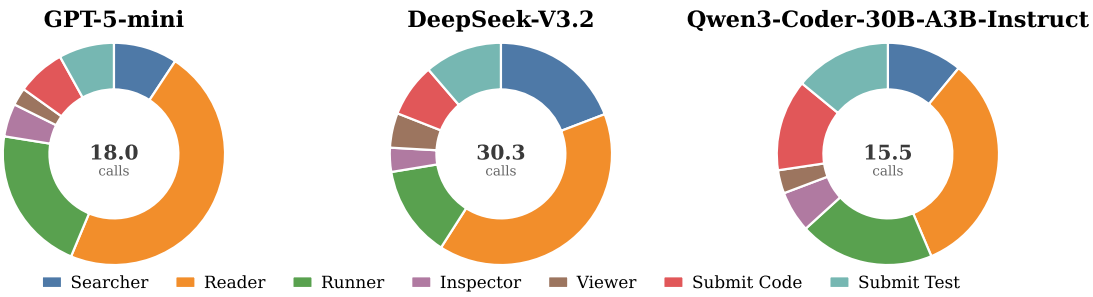


Figure 7: The tool usage statistics of TDD-Agent on three models.

| Model | Method | Wall-clock Time(s) |
|---|---|---|
| Qwen | mini-SWE-agent | 26.71 |
| | TDD-Agent-iter5 | 42.47 |
| | TDD-Agent-iter10 | 48.43 |

Table 10: Average wall-clock time for mini-SWE-agent and TDD-Agent on locally deployed Qwen for one task.

## C Detailed information on RepoEval

The detailed experimental results on RepoEval are presented in Table 7. Table 8 presents the information for each of the eight repositories used in RepoEval.

### Token Usage and Wall-Time

Table 9 presents the detailed token usage of TDD-Agent across iterations for three models on RepoEval. Since GPT and DeepSeek are accessed via API, cached token counts cannot be precisely determined. Qwen is deployed locally, allowing accurate measurement of cached tokens. We only report the cached tokens for Qwen. With caching enabled, TDD-Agent's token consumption would be further reduced.

The introduction of an iterative execution-refinement loop in TDD-Agent naturally incurs additional time overhead. Table 10 reports the wall-clock time of mini-SWE-agent and TDD-Agent. We report results only on Qwen, as it is deployed locally, thereby eliminating the effects of network latency and external service load. Although additional iterations introduce extra execution time, the resulting overhead remains acceptable in light of the substantial performance improvements.

### Tool Usage Statistics

Figure 7 presents the tool usage statistics of TDD-Agent across the three models. We exclude Early Terminator from the analysis because it can be invoked at most once per task.

### Failure Case Analysis

Qwen exhibits a relatively larger proportion of unmatched failures in Figure 6. This is consistent with the analysis in Section 4.1, where Qwen sometimes terminates even when its implementation does not pass the generated tests.

Table 11 presents the credit assignment problem rate in failure cases across three models, with the highest value being 8.10%, which does not exceed 10%.

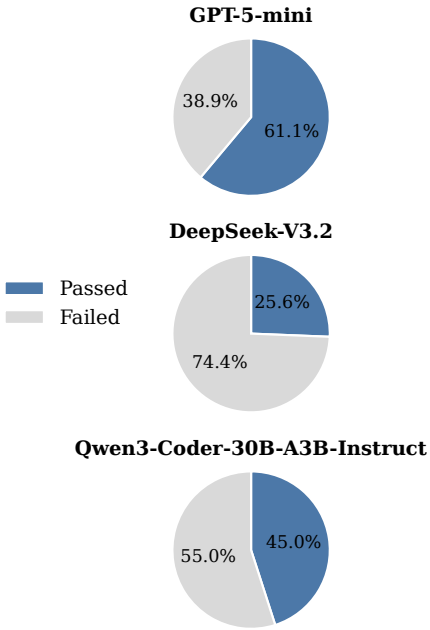


Figure 8: Generated-Tests Distribution in Matched Failures.

| Model | Credit Assignment Problem Rate |
|---|---|
| GPT | 3.03 % |
| DeepSeek | 4.76 % |
| Qwen | 8.10 % |

Table 11: The Credit Assignment Problem Rate in failure cases across three models.

Figure 8 presents the distribution of generated tests on ground-truth implementations within matched failures. If generated tests pass the ground-truth implementation, it indicates that the tests are consistent with the correct solution but are not sufficiently discriminative: they fail to distinguish the incorrect implementation from the expected behavior. In contrast, if a generated test suite fails the ground-truth implementation, this suggests that the test oracle itself is flawed, potentially due to incorrect assertions or misaligned expected outputs. Both types of issues appear across three models, showing that matched failures arise from a mixture of weak-but-valid tests and invalid tests. These findings suggest that future work should focus on stronger test-oracle construction, mutation-guided test expansion, and uncertainty-aware termination criteria to reduce false-positive verification.

## D Tool Implementation Details

TDD-Agent is implemented as a function-calling agent whose tool invocations are parsed and compiled into a small set of command-based actions.

Each tool is exposed to the model through a JSON schema specifying the tool name, description, argument types, and required fields. For every model response, the framework requires at least one valid tool call.

**Context Inspection.** `Directory Viewer` lists the contents of a directory, optionally recursively. To avoid exposing irrelevant or excessively large metadata, it always excludes `.git` directory, hides hidden files by default, sorts directory and file names deterministically, and caps the number of returned entries by a configurable `max_results` parameter. `File Reader` reads either an entire file prefix or a user-specified inclusive line range. Line numbers are 1-based, invalid ranges are rejected, and each call is capped at 200 lines to prevent the model from receiving excessively long file contents in a single observation. `Structure Inspector` is handled by the runtime and returns a lightweight structural summary of a Python file, including classes, function or method signatures, and line numbers, while omitting full implementations. `Code Searcher` provides repository-wide lexical search while avoiding unrestricted shell access. The model may issue raw Linux-style search commands, but the parser enforces three restrictions before execution. First, dangerous shell metacharacters and constructs, including command separators, redirection operators, subshells, and backticks, are rejected. Second, the pipe symbol is the only allowed command-composition operator. Third, after tokenization with `shlex`, every command appearing either at the beginning of the command or immediately after a pipe must belong to a fixed allowlist: `find`, `grep`, `egrep`, `fgrep`, `xargs`, `head`, `tail`, `cat`, `less`, `wc`, `sort`, `uniq`, `awk`, and `cut`. If a working directory is specified, the framework changes into that directory using shell-quoted paths before executing the validated command. This gives the agent enough flexibility for code search while preventing arbitrary command execution.

**Artifact Submission and Test Runner.** `Artifact Submitter` records the tests and implementation generated by the LLM. Specifically, submitted tests are written directly to `test_by_agent.py`, whereas submitted implementation code is used to replace the corresponding target code in the original repository file. `Test Runner` takes no arguments and executes only the most recently submitted generated test file, using `pytest`. During prediction, the repository's original hidden evaluation tests are never exposed to the agent and are not executable through `Test Runner`. They are used only after prediction for final evaluation. After each call to `Test Runner`, the resulting pytest report is appended to the conversation context and used as execution feedback for the next refinement step. The model may then revise the tests, revise the implementation, inspect more context, or call `Early Terminator`. The framework terminates when `Early Terminator` is called or when the predefined maximum number of refinement rounds is reached. In our main experiments, this maximum is set to 10.

## E All Prompts Used in Experiments

Table 12 and Table 13 present the prompts used in the function-level experiments and ablation studies.

For repository-level experiments, Table 14 presents the prompt for the full TDD-Agent setting, while Table 15, Table 16, and Table 17 present the prompts for the Vanilla, Reflect, and Single-track variants, respectively.

## F Case Study

Table 18 and Table 19 present an example problem and the corresponding model response for TDD-prompt in the function-level experiment.

Table 20 and Table 21 present an example of the generated tests and implementation produced by TDD-Agent in the repository-level experiment.

Table 12: Prompt for TDD-prompt on LiveCodeBench.

**# System**
You are an expert Python programmer. Given a programming problem, you must follow a strict thinking process before writing the final implementation.

**# User**
# Problem Description:

{description}

```python
{starter_code}
```

Please solve the problem by strictly following this process:
1. Overview: Identify the core logic, I/O format, constraints and edge cases.
2. Tests: Design 3-5 concrete test cases covering basic and edge scenarios.
- Use small-scale inputs to avoid miscalculation.
- You MUST briefly derive the correct expected output step-by-step.
3. Algorithmic Strategy: Briefly outline the optimal approach and its time/space complexity.
4. Implementation: Write the final Python code inside a ```python ``` block.

Table 13: Prompt for Ablation Study on LiveCodeBench.

**# System**
You are an expert Python programmer. Given a programming problem, you must follow a strict thinking process before writing the final implementation.

**# User**
# Problem Description:

{prompt}

```python
{starter_code}
```

Please solve the problem by strictly following this process:
1. Overview: Identify the core logic, I/O format, constraints and edge cases.
2. Algorithmic Strategy: Briefly outline the optimal approach and its time/space complexity.
3. Implementation: Write the final Python code inside a ```python ``` block.

Table 14: Prompt for TDD-Agent on RepoEval.

**# System**
You're a software engineer working inside an existing Python codebase rooted at /testbed.
Your task is to write test cases and complete the implementation of a target function whose implementation is currently missing.
You can use these tools:
1. `read_files`
2. `search_command`
3. `inspect_structure`
4. `list_files`
5. `submit_implementation`
6. `submit_tests`
7. `run_tests`
8. `finish`
Each of your response SHOULD include reasoning text explaining what you're doing
Each of your response MUST include AT LEAST ONE tool call.

**# User**
# Task Instructions

## Workflow

1. Inspect the target file and related code with `inspect_structure`, `read_files`, `search_command` and `list_files`
2. Create pytest test cases for the target function and submit with `submit_tests`
   - Focus on the canonical usage rather than obscure edge cases or stress testing
   - Your submitted tests will be saved as `test_by_agent.py` in the same directory as the target function file path
   - It will be later executed from the project root with `python -m pytest <path_to_target_dir>/test_by_agent.py`
3. Complete the implementation of the target function and submit with `submit_implementation`
   - Submit ONLY the implementation of the target function including the function signature but WITHOUT any imports or code outside the target function definition
   - DO NOT include anything else like any imports or contextual code
   - `submit_implementation` will directly OVERWRITE the target function in the original file with your submitted code
4. After submitting both your implementation and tests, validate with `run_tests`
   - Before you call `run_tests`, make sure both your implementation and tests are submitted
   - Calling `run_tests` will execute your latest submitted tests and implementation by running `python -m pytest <path_to_target_dir>/test_by_agent.py` command from the project root
5. Iteratively analyze the execution results, refine your tests and implementation, re-submit and re-run
6. Only when you are completely confident about both your tests and your implementation should you finish your task with `finish`

**# Execution Succeeds Prompt**
Your last execution was successful.
Now Reflect on :
   -whether the tests you wrote are sufficiently comprehensive and whether they cover all expected behaviors of the target function.
   -whether your implementation has any remaining issues and whether it fully realizes the expected behavior of the target function.
Only when you are completely confident about your implementation and tests, should you call `finish` to end your task.
Note:
   1. `run_tests` ONLY executes your latest submitted implementation and test cases.
   2. Running the project's pre-existing test suite is not allowed. Do not attempt to use `run_tests` to execute the repository's native test suite.
   3. Before you call `run_tests` again, please ensure that you have resubmitted at least one of tests and implementation.
   4. Rerunning `run_tests` without any new submissions is meaningless and will only yield identical results.
   5. Do not resubmit identical code before calling `finish`. The system will automatically save the exact version from your most recent `run_tests` execution.

**# Execution Fails Prompt**
Your last execution failed.
Carefully analyze the reasons for failure:
   -Is the implementation incorrect or incomplete?
   -Are the tests you wrote insufficient or incorrect?
Continue calling tools to refine your implementation and tests, submit and validate with `run_tests`.

Table 15: Prompt for Ablation Study on RepoEval: Vanilla.

**# System**
You're a software engineer working inside an existing Python codebase rooted at /testbed.
Your task is to write test cases and complete the implementation of a target function whose implementation is currently missing.
You can use these tools:
1. `read_files`
2. `search_command`
3. `inspect_structure`
4. `list_files`
5. `submit_implementation`
6. `finish`
Each of your response SHOULD include reasoning text explaining what you're doing
Each of your response MUST include AT LEAST ONE tool call.

**# User**
# Task Instructions

## Workflow

1. Inspect the target file and related code with `inspect_structure`, `read_files`, `search_command` and `list_files`
2. Complete the implementation of the target function and submit with `submit_implementation`
   - Submit ONLY the implementation of the target function including the function signature but WITHOUT any imports or code outside the target function definition
   - DO NOT include anything else like any imports or contextual code
   - `submit_implementation` will directly OVERWRITE the target function in the original file with your submitted code
3. Finish your task with `finish`

Table 16: Prompt for Ablation Study on RepoEval: Reflect.

**# System**
You're a software engineer working inside an existing Python codebase rooted at /testbed.
Your task is to write test cases and complete the implementation of a target function whose implementation is currently missing.
You can use these tools:
1. `read_files`
2. `search_command`
3. `inspect_structure`
4. `list_files`
5. `submit_implementation`
6. `finish`
Each of your response SHOULD include reasoning text explaining what you're doing
Each of your response MUST include AT LEAST ONE tool call.

**# User**
# Task Instructions

## Workflow

1. Inspect the target file and related code with `inspect_structure`, `read_files`, `search_command` and `list_files`
2. Complete the implementation of the target function and submit with `submit_implementation`
   - Submit ONLY the implementation of the target function including the function signature but WITHOUT any imports or code outside the target function definition
   - DO NOT include anything else like any imports or contextual code
   - `submit_implementation` will directly OVERWRITE the target function in the original file with your submitted code
3. Iteratively reflect on your implementation, analyze whether there are any remaining issues, refine your implementation and re-submit
4. Only when you are completely confident about your implementation should you finish your task with `finish`

Table 17: Prompt for Ablation Study on RepoEval: Single-track.

**# System**
You're a software engineer working inside an existing Python codebase rooted at /testbed.
Your task is to write test cases and complete the implementation of a target function whose implementation is currently missing.
You can use these tools:
1. `read_files`
2. `search_command`
3. `inspect_structure`
4. `list_files`
5. `submit_implementation`
6. `submit_tests`
7. `run_tests`
8. `finish`
Each of your response SHOULD include reasoning text explaining what you're doing
Each of your response MUST include AT LEAST ONE tool call.

**# User**
# Task Instructions

## Workflow

1. Inspect the target file and related code with `inspect_structure`, `read_files`, `search_command` and `list_files`
2. Create pytest test cases for the target function and submit with `submit_tests`
   - Focus on the canonical usage rather than obscure edge cases or stress testing
   - Your submitted tests will be saved as `test_by_agent.py` in the same directory as the target function file path
   - It will be later executed from the project root with `python -m pytest <path_to_target_dir>/test_by_agent.py`
3. Complete the implementation of the target function and submit with `submit_implementation`
   - Submit ONLY the implementation of the target function including the function signature but WITHOUT any imports or code outside the target function definition
   - DO NOT include anything else like any imports or contextual code
   - `submit_implementation` will directly OVERWRITE the target function in the original file with your submitted code
4. After submitting both your implementation and tests, validate with `run_tests`
   - Before you call `run_tests`, make sure both your implementation and tests are submitted
   - Calling `run_tests` will execute your submitted tests and your latest submitted implementation by running `python -m pytest <path_to_target_dir>/test_by_agent.py` command from the project root
5. Iteratively analyze the execution results, refine your implementation, re-submit and re-run
6. Only when you are completely confident about both your implementation should you finish your task with `finish`

**# Execution Succeeds Prompt**
Your last execution was successful.
Now Reflect on whether your implementation has any remaining issues and whether it fully realizes the expected behavior of the target function.
Only when you are completely confident about your implementation, should you call `finish` to end your task.
Note:
   1. `run_tests` ONLY executes your latest submitted implementation.
   2. Running the project's pre-existing test suite is not allowed. Do not attempt to use `run_tests` to execute the repository's native test suite.
   3. Before you call `run_tests` again, please ensure that you have resubmitted your implementation.
   4. Rerunning `run_tests` without any new submissions is meaningless and will only yield identical results.
   5. Do not resubmit identical code before calling `finish`. The system will automatically save the exact version from your most recent `run_tests` execution.

**# Execution Fails Prompt**
Your last execution failed.
Carefully analyze the reasons for failure:
   -Is the implementation incorrect or incomplete?
Continue calling tools to refine your implementation, submit and validate with `run_tests`.

Table 18: Case Study: An Example Problem from LiveCodeBench and TDD-Prompt's Final Answer.

**Problem Description:**

You are given an integer array enemyEnergies denoting the energy values of various enemies.
You are also given an integer currentEnergy denoting the amount of energy you have initially.
You start with 0 points, and all the enemies are unmarked initially.
You can perform either of the following operations zero or multiple times to gain points:

Choose an unmarked enemy, i, such that currentEnergy >= enemyEnergies[i]. By choosing this option:

You gain 1 point.
Your energy is reduced by the enemy's energy, i.e. currentEnergy = currentEnergy - enemyEnergies[i].
If you have at least 1 point, you can choose an unmarked enemy, i. By choosing this option:
Your energy increases by the enemy's energy, i.e. currentEnergy = currentEnergy + enemyEnergies[i].
The enemy i is marked.
Return an integer denoting the maximum points you can get in the end by optimally performing operations.

Example 1:
Input: enemyEnergies = [3,2,2], currentEnergy = 2
Output: 3
Explanation:
The following operations can be performed to get 3 points, which is the maximum:
First operation on enemy 1: points increases by 1, and currentEnergy decreases by 2. So, points = 1, and currentEnergy = 0.
Second operation on enemy 0: currentEnergy increases by 3, and enemy 0 is marked. So, points = 1, currentEnergy = 3, and marked enemies = [0].
First operation on enemy 2: points increases by 1, and currentEnergy decreases by 2. So, points = 2, currentEnergy = 1, and marked enemies = [0].
Second operation on enemy 2: currentEnergy increases by 2, and enemy 2 is marked. So, points = 2, currentEnergy = 3, and marked enemies = [0, 2].
First operation on enemy 1: points increases by 1, and currentEnergy decreases by 2. So, points = 3, currentEnergy = 1, and marked enemies = [0, 2].

Example 2:
Input: enemyEnergies = [2], currentEnergy = 10
Output: 5
Explanation:
Performing the first operation 5 times on enemy 0 results in the maximum number of points.

Constraints:
1 <= enemyEnergies$\dot{l}$ength <= $10^5$
1 <= enemyEnergies[i] <= $10^9$
0 <= currentEnergy <= $10^9$

**Final Answer:**

```
class Solution:
    def maximumPoints(self, enemyEnergies: List[int], currentEnergy: int) :
        if not enemyEnergies:
            return 0
        min_e = min(enemyEnergies)
        if currentEnergy < min_e:
            return 0
        total_sum = sum(enemyEnergies)
        return (currentEnergy + total_sum - min_e) // min_e
```

Table 19: Case Study: A Response Example Generated by TDD-Prompt.

```
1. Overview
- Core logic: You can repeatedly "attack" an unmarked enemy i (if currentEnergy
    >= e_i) to gain +1 point and consume e_i energy, and you can "mark" any
    unmarked enemy (only if you already have >=1 point) to gain e_i energy and
    prevent future attacks on that enemy. Marking does not cost points. The
    optimal strategy is:
  - Use one attack on a cheapest enemy (minimal e) to get the required 1 point
      to unlock marking.
  - Then mark all other enemies to collect their energies (marking never costs
      points, only prevents attacking those particular enemies later).
  - Finally, spend the accumulated energy attacking the cheapest enemy as many
      times as possible.
- I/O: Implement Solution.maximumPoints(enemyEnergies: List[int], currentEnergy:
     int) -> int.
- Constraints: n up to 1e5, energies and currentEnergy up to 1e9. Use O(n) time
    and 64-bit integers when summing.
- Edge cases:
  - If currentEnergy < min(enemyEnergies): can't get the first point -> answer
      0.
  - Single enemy case handled naturally.
  - Large sums: use Python ints (unbounded)

2. Tests (each with step-by-step expected output)
Test A: enemyEnergies = [3, 2, 2], currentEnergy = 2
- min = 2, sum = 7. current >= min so:
  - Attack min once: energy = 2 - 2 = 0, points = 1.
  - Mark the other enemies (2 and 3): energy = 0 + (2 + 3) = 5.
  - Attack min (2) floor(5 / 2) = 2 more times.
  - Total points = 1 + 2 = 3.
- Expected output: 3
Test B: enemyEnergies = [2], currentEnergy = 10
- min = 2, sum = 2. current >= min:
  - Attack once: energy = 8, points = 1. No others to mark.
  - Attack min floor(8 / 2) = 4 more times.
  - Total = 1 + 4 = 5.
- Expected output: 5
Test C: enemyEnergies = [5, 10, 3], currentEnergy = 3
- min = 3, sum = 18. current >= min:
  - Attack min once: energy = 0, points = 1.
  - Mark others (5 + 10): energy = 0 + 15 = 15.
  - Attack min (3) floor(15 / 3) = 5 more times.
  - Total = 1 + 5 = 6.
- Expected output: 6
Test D (edge: can't start): enemyEnergies = [4, 7, 9], currentEnergy = 3
- min = 4. current < min => cannot perform any attack and cannot mark.
- Expected output: 0

3. Algorithmic Strategy
- Let min_e = min(enemyEnergies) and sum_all = sum(enemyEnergies).
- If currentEnergy < min_e: return 0.
- Perform one attack on the cheapest enemy (cost min_e) to get the required 1
    point.
- Mark all the other enemies and collect their energies: net energy after that =
     currentEnergy - min_e + (sum_all - min_e) = currentEnergy + sum_all - 2 *
    min_e.
- Use that energy to attack the cheapest enemy as many times as possible: floor(
    net_energy / min_e).
- Total points = 1 + floor(net_energy / min_e). This simplifies to (
    currentEnergy + sum_all - min_e) // min_e.
- Time complexity: O(n). Space complexity: O(1) extra space.

4. Implementation
```python
class Solution:
    def maximumPoints(self, enemyEnergies: List[int], currentEnergy: int) :
        ...
```
```

Table 20: Case Study: An Example of Generated Tests by TDD-Agent.

```python
import jax
import jax.numpy as jnp
import numpy as np
import numpyro
from numpyro import handlers
import numpyro.distributions as dist
from lightweight_mmm.core import priors
from lightweight_mmm.core.transformations import saturation

def test_hill_custom_prior_half_max_effective_concentration():
    """Test that custom prior for half_max_effective_concentration is used."""
    expected_value1, expected_value2 = 5.2, 7.56
    custom_priors = {priors.HALF_MAX_EFFECTIVE_CONCENTRATION:
            dist.Kumaraswamy(
                concentration1=expected_value1, concentration0=expected_value2)}
    data = jnp.ones((10, 5))
    trace_handler = handlers.trace(handlers.seed(saturation.hill, rng_seed=0))
    trace = trace_handler.get_trace(data=data, custom_priors=custom_priors)
    prior_name = priors.HALF_MAX_EFFECTIVE_CONCENTRATION
    used_distribution = trace[prior_name]["fn"]
    used_distribution = used_distribution.base_dist
    assert isinstance(used_distribution, dist.Kumaraswamy)
    assert used_distribution.concentration0 == expected_value2
    assert used_distribution.concentration1 == expected_value1

def test_hill_custom_prior_slope():
    """Test that custom prior for slope is used."""
    expected_value1, expected_value2 = 3.0, 4.0
    custom_priors = {priors.SLOPE:
            dist.Kumaraswamy(
                concentration1=expected_value1, concentration0=expected_value2)}
    data = jnp.ones((10, 5))
    trace_handler = handlers.trace(handlers.seed(saturation.hill, rng_seed=0))
    trace = trace_handler.get_trace(data=data, custom_priors=custom_priors)
    prior_name = priors.SLOPE
    used_distribution = trace[prior_name]["fn"]
    used_distribution = used_distribution.base_dist
    assert isinstance(used_distribution, dist.Kumaraswamy)
    assert used_distribution.concentration0 == expected_value2
    assert used_distribution.concentration1 == expected_value1

def test_hill_with_zero_data():
    """Test hill function with zero input data"""
    data = jnp.zeros((10, 3))
    def mock_model():
        output = saturation.hill(data=data, custom_priors={})
        return numpyro.deterministic("output", output)
    trace_handler = handlers.trace(handlers.seed(mock_model, rng_seed=0))
    trace = trace_handler.get_trace()
    output = trace["output"]["value"]
    # With zero data, hill function should return zeros
    assert jnp.allclose(output, 0, atol=1e-7)

def test_hill_shape_3d():
    """Test that hill function handles 3D input correctly."""
    data = jnp.ones((100, 3, 5))
    def mock_model():
        output = saturation.hill(data=data, custom_priors={})
        return numpyro.deterministic("output", output)
    trace_handler = handlers.trace(handlers.seed(mock_model, rng_seed=0))
    trace = trace_handler.get_trace()
    output = trace["output"]["value"]
    assert output.shape == data.shape
```

Table 21: Case Study: An Example of Generated Implementation by TDD-Agent.

```
def hill(
    data: jnp.ndarray,
    custom_priors: Mapping[str, dist.Distribution],
    *,
    prefix: str = "",
) -> jnp.ndarray:
  """Transforms the input data with the adstock and hill functions.

  Args:
    data: Media data to be transformed. It is expected to have 2 dims for
      national models and 3 for geo models.
    custom_priors: The custom priors we want the model to take instead of the
      default ones. The possible names of parameters for hill_adstock and
      exponent are "lag_weight", "half_max_effective_concentration" and "slope".
    prefix: Prefix to use in the variable name for Numpyro.

  Returns:
    The transformed media data.
  """
  default_priors = priors.get_default_priors()

  with numpyro.plate(
      name=f"{prefix}{priors.HALF_MAX_EFFECTIVE_CONCENTRATION}_plate",
      size=data.shape[1]):
    half_max_effective_concentration = numpyro.sample(
        name=f"{prefix}{priors.HALF_MAX_EFFECTIVE_CONCENTRATION}",
        fn=custom_priors.get(priors.HALF_MAX_EFFECTIVE_CONCENTRATION,
                             default_priors[priors.
                                 HALF_MAX_EFFECTIVE_CONCENTRATION]))

  with numpyro.plate(
      name=f"{prefix}{priors.SLOPE}_plate",
      size=data.shape[1]):
    slope = numpyro.sample(
        name=f"{prefix}{priors.SLOPE}",
        fn=custom_priors.get(priors.SLOPE, default_priors[priors.SLOPE]))

  if data.ndim == 3:
    half_max_effective_concentration = jnp.expand_dims(
        half_max_effective_concentration, axis=-1)
    slope = jnp.expand_dims(slope, axis=-1)

  return _hill(
      data=data,
      half_max_effective_concentration=half_max_effective_concentration,
      slope=slope)
```