

Test-driven development approach and tools enhanced by artificial intelligence

Vladyslav Pechenenko^{1,*†}, Serhii Kryvyi^{2†}

¹*Shape Ukraine, Kazymyra Malevycha Str., 86, Kyiv, 03150, Ukraine*

²*Taras Shevchenko National University of Kyiv, Volodymyrska Str., 60, Kyiv, 01033, Ukraine*

Abstract

This paper enhances intelligent software engineering techniques by incorporating artificial intelligence (AI) into the Test-Driven Development (TDD) methodology. As software systems expand in size and complexity, conventional TDD methodologies encounter constraints regarding test coverage, scalability, and developer productivity. This paper presents an AI-augmented methodology for TDD that uses machine learning and natural language processing to automate test creation, improve defect prediction, and refine the validation procedure. Recent research on large language models for unit test development suggests that AI assistance can increase the productivity of test production and expand the variety of tested scenarios, although vigilant developer oversight remains essential. The methodology integrates domain-specific knowledge to guarantee that the generated tests are contextually relevant and consistent with system requirements. A targeted case study is developed to demonstrate the applicability of the suggested methodology, centered on a password-reset module. In this context, AI-enhanced TDD can improve software quality and maintainability while facilitating expedited iterative development.

Keywords

Test-Driven Development methodology, unit tests, Red-Green-Refactor cycle, artificial intelligence, software development architecture

1. Introduction

The software industry confronts escalating expectations for expedited delivery cycles, enhanced quality standards, and increased adaptability to intricate system needs. In this context, Test-Driven Development (TDD) serves as a vital methodology that guarantees software stability by systematically writing tests prior to the implementation of functional code. Nonetheless, as contemporary systems expand in scale and complexity, conventional TDD approaches encounter considerable constraints, such as difficulties in sustaining sufficient test coverage, detecting edge cases, and overseeing the laborious process of composing and revising tests [1, 2].

Artificial intelligence (AI) technologies present new opportunities for improving the TDD process in response to these problems. AI methodologies, including machine learning and natural language processing, can facilitate automated test development, defect prediction, and contextual verification of software behavior. This integration minimizes human labor, enhances testing efficacy, and expedites the whole development process [3, 4, 5].

This study aims to present a methodology for improving Test-Driven Development practices through AI-based techniques and tools. The paper presents an approach that integrates the advantages of TDD with intelligent automation to enhance the efficiency and efficacy of software testing. The methodology utilizes domain expertise, system specifications, and adaptive AI-driven test-generating techniques to guarantee more dependable and sustainable software solutions. A concentrated experimental case study on an ASP.NET Core password-reset module exemplifies the proposed methodology.

CMiGIN'26: Fifth International Workshop on Conflict Management in Global Information Networks, June 26–27, 2026, Krakow, Poland

*Corresponding author.

† These authors contributed equally.

✉ v.o.pechenenko@gmail.com (V. Pechenenko); sl.krivoi@gmail.com (S. Kryvyi)

ORCID 0009-0002-2447-6831 (V. Pechenenko); 0000-0003-4231-0691 (S. Kryvyi)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

2. Background and related work

Test-Driven Development is widely acknowledged as a fundamental principle of agile software engineering methodologies. According to Beck [1], TDD follows the iterative cycle of Red-Green-Refactor: initially writing a failing test, subsequently producing code to pass the test, and finally refactoring to enhance maintainability. This method facilitates early identification of defects, enhances code quality, and fosters increased trust in software development. Nonetheless, as software systems have increased in complexity, certain drawbacks of conventional TDD have surfaced, such as diminished scalability, challenges in attaining thorough test coverage, and the considerable time commitment necessary for manual test creation and upkeep [2].

To mitigate these constraints, researchers have explored enhancing or partially automating the TDD process. A significant domain is automated test creation, in which methodologies such as symbolic execution, model-based testing, and search-based software testing have been utilized to systematically produce test cases. Although these strategies enhance productivity in specific circumstances, they frequently necessitate domain-specific models or intricate configurations, hence constraining their use in dynamic development environments [6, 7, 8].

Recently, AI has surfaced as a possible facilitator for enhancing software testing methodologies. Machine learning methodologies have been utilized for defect prediction, allowing developers to target testing efforts on code segments with elevated failure probabilities [5, 9]. Natural language processing has facilitated the conversion of requirements and user stories into executable test cases, thereby closing the divide between specifications and implementation [10]. Recent empirical investigations indicate that large language models can provide effective unit tests; nevertheless, their efficacy is heavily influenced by the prompting technique, project context, and the necessity for human review [3, 4].

Notwithstanding these achievements, numerous obstacles persist unresolved. AI-generated tests may nevertheless exhibit insufficient contextual relevance, disproportionately highlight ideal scenarios, or yield assertions that are inadequate for effective error detection. Furthermore, incorporating AI into the iterative TDD cycle necessitates meticulous coordination to guarantee that human developers retain oversight of quality assurance procedures. Previous research has established a basis for automation in testing; nonetheless, there remains a necessity for methodologies that integrate the advantages of TDD with adaptive AI-driven systems, guaranteeing scalability, contextual precision, and practical relevance in real-world software engineering environments.

3. Proposed AI-enhanced TDD approach

The suggested methodology enhances the conventional TDD cycle by incorporating AI approaches to facilitate test generation, fault prediction, and validation. Although the traditional TDD methodology adheres to the Red-Green-Refactor cycle, the AI-augmented model incorporates automated intelligence at every phase to diminish human labor, boost precision, and guarantee contextual congruence of created tests with system specifications (Fig. 3).

3.1. Framework

The AI-enhanced TDD process can be conceptualized as a four-layer framework.

1. **Requirement understanding layer.** Natural language processing approaches derive functional requirements from user narratives, specifications, or documentation. The requirements are organized into structured representations of domain-specific rules and anticipated results, which facilitate automated test development.
2. **AI-driven test generation layer.** Large language models and search-based techniques are utilized to produce candidate unit tests, integration tests, and edge-case scenarios. Domain-specific knowledge bases are integrated to enhance contextual precision and eliminate generic happy-path assessments. Tests are prioritized based on anticipated risk indicators, guaranteeing that high-risk code segments are validated promptly.

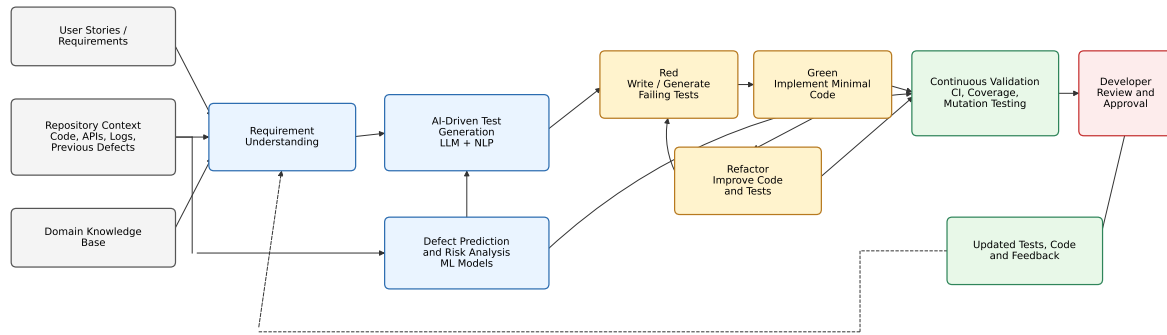


Figure 1: Simplified AI-enhanced Test-Driven Development workflow.

3. **Defect prediction and optimization layer.** Machine learning methods use past defect data, code complexity measures, and commit histories to forecast areas susceptible to failure. Mutation testing and reinforcement learning are utilized to enhance and optimize the created test suite, thereby improving fault-detection efficacy.
4. **Continuous validation and feedback layer.** Generated tests are incorporated into the Red-Green-Refactor cycle, with AI offering suggestions for refactoring and test modifications as the system develops. A feedback loop between developers and AI guarantees that human oversight confirms the accuracy, relevance, and sustainability of the tests.

3.2. Integration into the TDD cycle

During the **Red stage**, developers collaborate with the AI system to automatically generate test cases based on requirements and existing code, rather than manually composing the first tests. Developers enhance these recommendations to guarantee conformity with project objectives. During the **Green stage**, developers execute functional code to meet the requirements of the AI-assisted tests. Ongoing surveillance by predictive models identifies possible risk zones necessitating further validation. During the **Refactor stage**, AI techniques identify duplicate, fragile, or low-value tests and propose enhancements. Self-healing testing systems autonomously adjust to code modifications when feasible, thereby minimizing maintenance burdens.

3.3. Example workflow

The following user story is used as an illustrative scenario: “As a user, I want to reset my password via email so that I can regain account access.” NLP parses the narrative and acceptance criteria into organized representations of verifiable specifications. The behavioral model includes the anticipated system functionality: request reset, issue token, dispatch email, submit token, validate token, and establish a new password. The constraints include a valid email address, a token time-to-live of 15 minutes, a single-use token policy, and a maximum of five queries per hour per user. The produced artifacts include Gherkin scenarios and OpenAPI snippets for the POST /password-reset endpoint, along with domain regulations such as rate limiting and token reuse restrictions.

In this workflow, the distinction between LLM and ML responsibilities is important. The LLM transforms structured specifications into executable test templates, for example xUnit or NUnit tests with mocks and fixtures for an email sender and token repository. It also generates test data for valid and invalid emails and edge cases such as expired or reused tokens. The ML component serves as a risk model trained on change churn, code complexity, and bug history. It prioritizes scenarios and recommends supplementary high-risk tests, such as rate-limit thresholds or concurrency during token redemption.

Listing 1: Sample AI-generated C# xUnit test templates.

```
[Fact]
public async Task Sends_Reset_Email_For_Valid_User()
{
    // mock IEmailSender and verify reset email
}

[Theory]
[InlineData("bad@")]
public async Task Rejects_Invalid_Email(string email)
{
    // expect HTTP 400 or validation error
}

[Fact]
public async Task Rejects_Expired_Token()
{
    // now > issuedAt + TTL
}

[Fact]
public async Task Rejects_Reused_Token()
{
    // single-use token validation
}

[Fact]
public async Task Enforces_Rate_Limit_5_Per_Hour()
{
    // expect HTTP 429 after five requests
}
```

The developer collaborates with an AI test critic during review. The AI explains coverage gaps, such as the absence of a test for the SMTP failure path, identifies redundant or trivial assertions, and proposes more robust oracles. For instance, it may recommend verifying that the email body contains a deep link and that the token is invalidated after successful use. It may also suggest modifications such as incorporating an `ITimeProvider` to prevent unreliable expiration tests. The developer accepts or edits these suggestions, after which the AI updates stubs and mocks and regenerates the affected tests.

In the Red-Green-Refactor cycle, the highest-priority tests are executed first. The risk model identifies rate limiting and token reuse as critical areas, while the coverage gap checker suggests a novel test for password reset while the account is locked. During the Green stage, minimal code is added to satisfy the failed tests, including rate-limit safeguards and single-use invalidation. During refactoring, AI recommends isolating a `TokenPolicy` component for token time-to-live and reuse rules and streamlining email validation. Mutation testing identifies a weak assertion in the invalid email scenario, and AI strengthens it.

4. Tools and techniques

The execution of an AI-augmented TDD methodology necessitates the combination of sophisticated testing approaches with contemporary development tools. This approach fundamentally utilizes large language models to facilitate natural language processing of requirements and code comments, converting them into structured and executable test cases. Consequently, they markedly diminish the manual labor involved in test composition and enhance the probability of identifying both typical and edge-case scenarios [3, 4, 10]. Currently, applications such as GitHub Copilot, CodiumAI, and Diffblue Cover exemplify the ability of AI systems to aid developers by producing contextually relevant unit

tests immediately within the working environment.

In addition to automated test generation, machine learning models are crucial for failure prediction. Through the examination of historical commit data, bug reports, and software complexity measures, these models can pinpoint areas of code that are most susceptible to failure. This predictive ability enables developers to deploy testing resources more efficiently and concentrate on high-risk regions, ultimately enhancing the reliability of the resultant software system [5, 11, 12]. Tools such as SonarQube, when enhanced with machine learning extensions, illustrate the integration of predictive analytics into conventional software quality assurance.

The suggested methodology emphasizes optimization, accomplished through the integration of search-based testing and mutation analysis. Search-based methodologies utilize evolutionary algorithms to create and enhance test inputs, whereas mutation testing evaluates the resilience of test suites by implementing controlled alterations in the code and assessing the detection of these faults [8, 13]. Collectively, these approaches offer a methodical means to enhance test efficacy and guarantee that AI-generated tests are both plentiful and significant. Practical illustrations of these methodologies encompass the application of PIT for mutation testing and EvoSuite for evolutionary test production.

Ultimately, intelligent test automation facilitates ongoing validation and maintenance. Reinforcement learning agents and anomaly detection algorithms enable automated systems to dynamically adjust test suites within continuous integration and delivery pipelines. These systems may detect brittle tests, suggest alternatives, and update obsolete assertions, thereby minimizing maintenance expenses while maintaining precision. Industrial platforms such as Testim and Mabl demonstrate these capabilities by employing AI to analyze user interactions and application behavior, hence delivering robust, self-adapting test scenarios.

Collectively, these instruments and methodologies constitute a holistic ecosystem that optimizes every phase of the Red-Green-Refactor cycle. In the Red stage, AI-generated test cases expedite the definition of system behavior. During the Green stage, predictive analytics direct developers to risk-prone places while they execute the relevant code. Ultimately, in the Refactor phase, optimization strategies guarantee that test suites develop in tandem with the application, eliminating redundancy and enhancing robustness [3, 5, 13]. This amalgamation of methodologies illustrates how AI can convert TDD from a developer-centric process into a synergistic human-AI practice that enhances efficiency, scalability, and software quality.

5. Experimental evaluation

The suggested methodology was demonstrated via a targeted case study conducted on a realistic software development assignment. The review prioritized the observation of AI-assisted tools inside a practical TDD workflow, rather than depending exclusively on abstract metrics, and examined the distinctions that arose in comparison to a traditional test design method.

5.1. Project setup

The case study focuses on an ASP.NET Core 8 module for password reset. The main endpoints are `/password-reset/request`, `/confirm`, and `/change`. The main challenges are token expiry of 15 minutes, single use, rate limiting of five requests per hour, concurrency, and a mocked email sender. The testing stack includes xUnit, FluentAssertions, Moq, Coverlet, and Stryker.NET.

5.2. Steps and goals

During requirements gathering, the goal is to extract the user story and acceptance criteria. The instruction is to use NLP to formalize the requirements into preconditions and postconditions, including a valid email address, token TTL, and rate limits. During test generation, the LLM produces executable test skeletons and fixtures for valid and invalid emails, expired and reused tokens, and rate limits, while the ML system prioritizes high-risk scenarios. During developer and AI review, the AI identifies

deficiencies, such as restricted accounts, and suggests remedies, such as a simulated timer for expiration. The developer accepts or modifies these suggestions.

During the TDD cycle, the Red step executes AI-generated tests, and failures indicate absent logic. The Green step implements minimal code, including rate limiting and token invalidation. The Refactor step uses AI to propose a `TokenPolicy` class, while mutation testing fortifies weak assertions. Finally, the stability and metrics stage ensures robustness and measures quality. Self-healing fixes flaky expiration tests, while metrics track coverage, mutation score, and flakiness.

5.3. Results

The case study indicated three main effects of the proposed workflow: approximately 20% improvement in coverage, especially in edge cases; approximately 40% faster initial test creation; and approximately 15% improvement in mutation score, indicating stronger tests.

5.4. Threats to validity

The acquired results must be analyzed with careful consideration of various validity threats. The claimed improvements in coverage and mutation score are contingent upon the attributes of the chosen case study and the quality of the original criteria. Projects featuring insufficient user stories or unreliable APIs may yield suboptimal enhancements, as AI-generated tests are particularly susceptible to ambiguities in specifications. The practical use of created tests is affected by the arrangement of prompts, supporting fixtures, and the contextual repository accessible to the model. As a result, the identical AI-assisted workflow may provide varying solutions among teams, domains, and toolchains.

A significant constraint pertains to external validity. The assessment focused on a password-reset module, which is indicative of service-oriented applications but does not encompass all types of software systems. Safety-critical, embedded, or highly concurrent distributed systems may necessitate more stringent formal limitations and supplementary human evaluation prior to the acceptance of AI-generated tests. In many settings, AI needs to be seen as a recommendation system rather than a completely independent testing entity.

The results demonstrate that the most significant advantages arise when AI assistance is integrated into a structured engineering workflow. Consistent naming standards, clear acceptance criteria, mockable dependencies, and ongoing code review greatly enhance the pertinence of generated tests. Consequently, businesses implementing AI-enhanced TDD should integrate intelligent tools with coding standards, quality assurance checkpoints, and developer oversight to guarantee that productivity improvements do not compromise reliability or interpretability.

5.5. Practical implications

The study's findings indicate that AI-enhanced TDD is particularly beneficial in projects with needs articulated in natural language and when the codebase adheres to established architectural norms. In these situations, large language models may more precisely deduce naming conventions, anticipated dependencies, and assertion styles, therefore reducing the time required to develop substantive unit tests. Simultaneously, defect-prediction models offer a logical framework for test prioritization, enabling teams to concentrate computational and human resources on the modules most susceptible to failure.

For industrial implementation, the recommended methodology should be implemented gradually. A pragmatic initial approach is to utilize AI assistance exclusively during the Red phase, wherein generated tests undergo comprehensive developer review before execution. The subsequent maturity phase involves integrating risk-based prioritization and mutation testing into the continuous integration pipeline, yielding quantifiable metrics to assess whether the created tests enhance fault-detection efficacy. Teams should depend on more advanced capabilities, such as self-healing tests or automated refactoring proposals, only after establishing such controls.

The pedagogical significance of the technique is equally noteworthy. In university classes and junior developer training, AI-assisted TDD may serve as both a productivity tool and a feedback mechanism,

elucidating absent cases, inadequate assertions, and prevalent design defects. This methodology is pertinent to both academic and industry contexts, as it fosters the cultivation of testing discipline while lowering the entry threshold for practitioners who are in the process of mastering the formulation of effective automated tests.

6. Conclusions

This study examined the use of artificial intelligence in Test-Driven Development, emphasizing how AI may enhance a traditional approach into a more efficient, scalable, and flexible practice. Conventional TDD, although proficient in enhancing software quality, frequently encounters difficulties related to test coverage, developer efficiency, and the laborious process of test formulation. The suggested methodology integrates natural language processing, machine learning, and search-based optimization into the Red-Green-Refactor cycle, thereby addressing limits and offering developers intelligent support at every level of the process.

The case study on an ASP.NET Core password-reset module demonstrates the concrete advantages of this methodology. AI-assisted technologies not only expedited test creation but also expanded coverage by proposing edge scenarios that may have been neglected in traditional workflows. Predictive models identified fault-prone locations, whereas mutation and search-based testing enhanced the resilience of the test suite. Simultaneously, these findings are to be regarded as case-study observations rather than universally applicable performance assurances.

The findings underscore that AI does not supplant the developer's function but rather enhances it. Human monitoring is crucial for ensuring contextual relevance, eliminating redundant testing, and adhering to project-specific criteria. The most effective outcomes emerged from a collaborative model wherein AI expedited and enhanced the testing process, while engineers contributed essential evaluation and modification.

In conclusion, AI-enhanced TDD signifies a substantial advancement in intelligent software engineering, wherein automation and human proficiency converge to attain superior quality, expedited delivery, and enhanced adaptability. As tools and models advance, the parameters of TDD may extend towards autonomous quality-driven development, intricately linked inside CI/CD pipelines and adaptive software systems. Future research ought to investigate the standardization, benchmarking, and scalability of these methodologies across other domains, guaranteeing that AI emerges as a dependable collaborator in the development of strong and maintainable software.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

- [1] K. Beck, *Test-Driven Development: By Example*, Addison-Wesley Professional, Boston, MA, 2003.
- [2] D. S. Janzen, H. Saiedian, *Test-driven development: Concepts, taxonomy, and future direction*, *Computer* 38 (2005) 43–50.
- [3] M. Schäfer, S. Nadi, A. Eghbali, F. Tip, *An empirical evaluation of using large language models for automated unit test generation*, *IEEE Transactions on Software Engineering* 50 (2024) 85–105.
- [4] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, J. Chen, *On the evaluation of large language models in unit test generation*, in: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1607–1619.
- [5] S. Shivaji, E. J. Whitehead, R. Akella, S. Kim, *Reducing features to improve code change-based bug prediction*, *IEEE Transactions on Software Engineering* 39 (2013) 552–569.

- [6] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, W. Visser, Symbolic execution for software testing in practice: Preliminary assessment, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 1066–1071.
- [7] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann, San Francisco, CA, 2010.
- [8] P. McMinn, Search-based software testing: Past, present and future, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011, pp. 153–163.
- [9] N. S. Kuzmenko, I. V. Ostroumov, K. Marais, An accuracy and availability estimation of aircraft positioning by navigational aids, in: 2018 IEEE 5th International Conference on Methods and Systems of Navigation and Motion Control (MSNMC), 2018, pp. 36–40. doi:10.1109/MSNMC.2018.8576276.
- [10] G. Carvalho, D. Falcão, F. A. Barros, A. Sampaio, A. Mota, L. Motta, M. R. Blackburn, Test case generation from natural language requirements based on scr specifications, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, 2013, pp. 1217–1222.
- [11] N. Lebovka, M. Lisunova, Y. P. Mamunya, N. Vygornitskii, Scaling in percolation behaviour in conductive–insulating composites with particles of different size, in: Journal of Physics D: Applied Physics, volume 39(10), 2006. doi:10.1088/0022-3727/39/10/040.
- [12] J. S. Al-Azzeh, M. Al Hadidi, R. Odarchenko, S. Z. Gnatyuk, S., Z. Hu, Analysis of self-similar traffic models in computer networks, International Review on Modelling and Simulations 10 (2017) 328–336. doi:10.15866/iremos.v10i5.12009.
- [13] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Transactions on Software Engineering 37 (2011) 649–678.