International Journal of Sustainability and Innovation in Engineering (IJSIE)
https://www.doi.org/10.56830/IJSIE2026
Vol. 4 2026

From Bugs to Brilliance: The Role of TDD in Embedded Software Engineering

https://www.doi.org/10.56830/IJSIE202621

Kishore Ranjan
Independent Researcher, Trumbull, United States
Email: kishoreranjan@gmail.com

Received: 23 July, 2026, Accepted: 31st October, 2026, Published: 18th November, 2026

Abstract:

Software development and qualification strategy for embedded systems largely depend on the system complexity and development & maintenance cost. This paper presents an overview of research on Test-Driven Development (TDD), focusing on software quality, productivity, and embedded systems engineering. Previous studies report mixed findings regarding TDD's effects on development productivity and external software quality. This paper discusses these findings and examines their implications for large-scale, complex, and safety-critical embedded software. It also considers emerging opportunities and limitations associated with using generative AI tools to support TDD. The paper introduces the main principles of TDD and discusses reported effects on software quality and productivity. It also examines how generative AI may support test and code generation. However, existing findings do not establish that TDD consistently improves quality or reduces productivity across all development contexts. Its suitability for complex embedded systems therefore requires context-specific evaluation.

Index Terms: Test-Driven Development (TDD); Embedded Systems; Software Quality; Test Automation; Lithography Machine

1. Introduction

Our lives are increasingly surrounded and dependent on embedded software systems. While there are a plethora of software development and verification techniques, software failures continue to grow in number. A 2022 study found that software failures cost US companies a staggering $2.41 trillion annually, up from $2.08 trillion in 2020 (Consortium for Information and Software Quality, 2023).

The goal of this paper is to evaluating an important aspect of Test Driven Development (TDD) where programmers write functional tests before the corresponding production code and how it can be helpful for a large and complex embedded software systems. In this exploratory work, a thorough analysis of research done across industries and academia in last two decades are done to explore how Test-Driven Development (TDD) can affects the productivity and quality of external code and how it can influence the Embedded System Software development, especially for a large code base and complex systems.

2. Background and Motivation

The concept of TDD is not new and has been under discussion for more than two decades (Beck, 1999). Many exploratory works are conducted with developers and researchers to explore the merits and short fall of TDD. Advocates for test-driven development claim that TDD produces code that's simpler, more cohesive, and less coupled than code developed in a more traditional test-last way. Support for TDD is growing in many development contexts but does it really helps improving the software quality (Ranjan et al., 2025) becomes always questionable.

This paper provides a narrative overview of selected research on TDD, focusing on external software quality and development productivity. It discusses findings from empirical studies, meta-analyses, and research concerning embedded software development. The paper does not claim to be a systematic literature review because it does not follow a predefined search strategy, documented selection criteria, or formal study-quality assessment.

The main purpose of this overview is to help practitioners understand the potential benefits and limitations of TDD in large-scale, complex, and safety-critical embedded software projects. It highlights considerations that engineering teams should evaluate before adopting TDD within their development and qualification processes.

3. Insights about TDD

3.1. TDD: An Introduction

Test-driven development (TDD) is not a new concept and has been practiced since long time (Beck, 1999, 2003; Cui, 2024; Dohmke, 2008; Erdogmus et al., 2005; Rafique & Mišić, 2013). It is an agile software development approach with test-first programming concepts. TDD is commonly considered as test-first development, in which unit tests are written before the production code (Matcha & Singh, 2025; Ranjan et al., 2025), which includes refactoring and restructuring a piece of code that passes the tests in order to reduce its complexity and improve its clarity, understandability, extendibility, and/or maintainability (Astels, 2003). TDD is all about improving the software design, by introducing tests at early phase.

TDD is often described with the so-called "red-green-refactor cycle" (Andrea, 2009):

- **Red** – Write a failing test (since the code doesn't exist yet).
- **Green** – Write minimal code to pass the test.
- **Refactor** – Improve the code while ensuring tests still pass.

It consists of the following steps:

1. Design and add a test.
2. Run all tests and see the new one fail (red).
3. Add enough implementation code to satisfy the new test.
4. Run all tests, repeat 3 if necessary until all tests pass (green).
5. Occasionally refactor to improve code structure.
6. Run all tests after refactoring to ensure all tests pass.

As an example (Piya & Sullivan, 2024), consider building a calculator program and starting with a test that adds 2 and 3 together. The first iteration of TDD would produce:

```
def test_add_positives
    assert add(2,3) == 5
def add(x, y):
    return 5
```

The minimal code the user needs to create in order for this test to pass is to simply have the add method directly return "5." In the next iteration, the user would pick another test for add:

```
def test_add_positives
    assert add(2,3) == 5
def test_add_mixed
    assert add(-2,3) == 1
def add(x, y):
    return x + y
```

In order to make the new and original test pass, the user will now update the function to add the two integers, x and y, together.

3.2. Development Flow

Figure 1a illustrates a traditional test-last flow of development. This process involves significant effort in specifying the system architecture and design before any significant software development. Such an approach does not preclude some programming to explore a prototype or prove a concept, but it assumes that no significant production software is constructed without a detailed design.

Figure 1b illustrates the test-first development flow. In this approach, the project identifies some high-level architecture early, but that design doesn't proceed to a detailed level. Instead, the test-first process of writing unit tests (Janzen & Saiedian, 2008) and constructing the units in short, rapid iterations allows the design to emerge and evolve

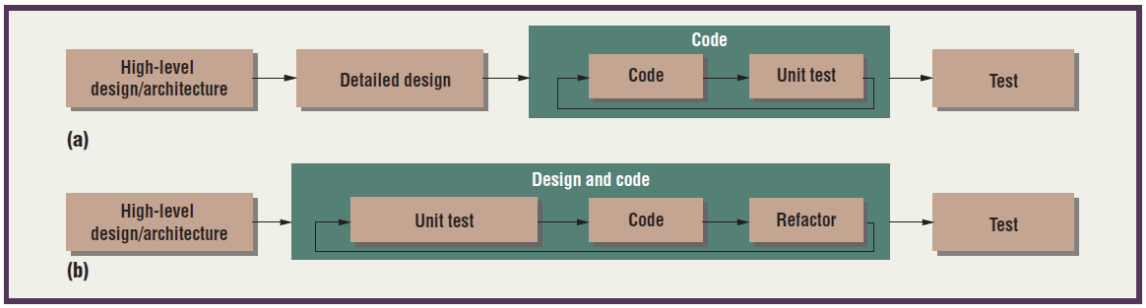


Figure 1. Development flow: (a) traditional test-last and (b) test-driven development/test-first flow (adapted from Janzen & Saiedian, 2008)

3.3. Key Highlights

TDD has following key advantages (Ranjan et al., 2025):

- Fast Feedback Loop: Tests provide the programmer with instant feedback as to whether new functionality has been implemented as intended and whether it interferes with old functionality.
- Task-orientation: Tests drive the coding activity, encouraging the programmer to decompose the problem into manageable, formalized programming tasks, helping to maintain focus and providing steady, measurable progress.
- Enhanced "Low-level" design: Tests provide the context in which low-level design decisions are made, such as which classes and methods to create, how they will be named, what interfaces they will possess, and how they will be used.
- Elevated Quality checks: Having up-to-date tests in place ensures a certain level of quality, maintained by frequently running the tests.

The deeper insight uncovered following unfortunate misconceptions exposed in a research survey (Janzen & Saiedian, 2008), with professional development teams, who were using TDD and willing to participate in the study interviewed representatives from four reputable Fortune 500 companies who claimed to be using TDD:

- *TDD equals automated testing*. Because TDD has helped propel automated testing to the forefront, many seem to think that TDD is only about writing automated tests.
- *TDD means write all tests first*. Some developers thinks that TDD involved writing the tests (all the tests) first, rather than using the short, rapid test-code iterations of TDD.

Since TDD essentially entails advancing test development into code development, it will be simpler to implement if you already have off-target unit tests. Because it moves receiving feedback to the left on the development timetable, this is known as "shift left". To enable test isolation, we might need to modify our software in addition to spending time building unit test

infrastructure. This endeavor has the advantage of exposing the code to testing (Dojo Five, n.d.).

4. Research and Survey Results

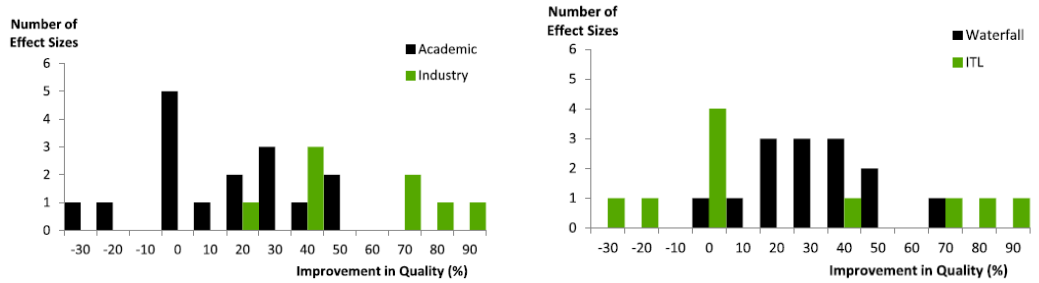
The objective of TDD was to bring improvements in code quality and productivity. However, research studies investigating the effectiveness of TDD could not give very distinct and conclusive results; in fact, all possible outcomes—positive, negative, and neutral—have been reported for both quality and productivity improvements obtained with TDD.

4.1. Experimental Results

Several empirical studies and research syntheses have examined the effects of TDD on software development and qualification. This overview discusses selected publications relevant to software quality, productivity, code complexity, and embedded systems. It does not claim exhaustive coverage of the literature. There are two key driving factors for industrial projects, "Quality" & "Productivity". Rafique and Mišić (2013) conducted a meta-analysis examining TDD's effects on external software quality and productivity. Their findings provide useful evidence for discussing these outcomes, but they do not establish that identical effects occur across all projects. The results should therefore be interpreted alongside differences in study design, project context, and implementation practices.

4.1.1. Impact on Software Quality

In a meta-analysis of data research (Rafique & Mišić, 2013), as highlighted in Figure 2, it is witnessed that the impact of TDD in terms of improving software quality is more visible in industrial projects than the academic projects.

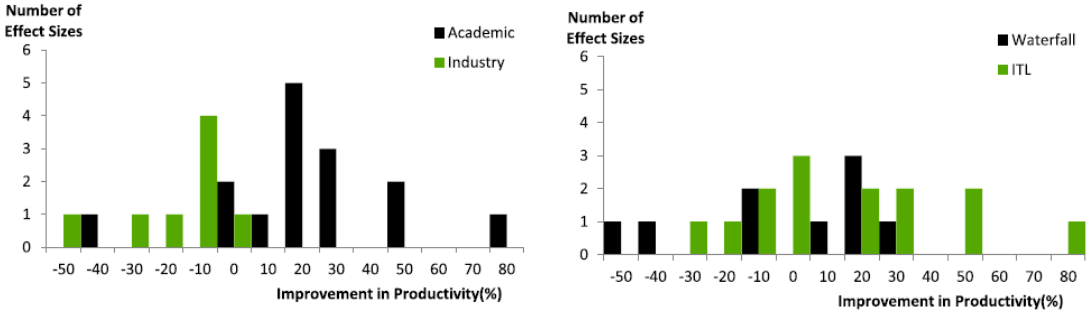


(a) Distribution by Academic/Industry subgroups (b) Distribution by Waterfall/ITL subgroups

Figure 2. Impact on quality (adapted from Rafique & Mišić, 2013)

4.1.2. Impact on Productivity

In the same study (Rafique & Mišić, 2013), the influence of TDD in productivity is not clearly visible in industrial projects as seen in Figure 3.



(a) Distribution by Academic/Industry subgroups (b) Distribution by Waterfall/ITL subgroups

Figure 3. Impact on productivity (adapted from Rafique & Mišić, 2013)

4.1.3. Impact on Code Coverage and Complexity

Another interesting parameter to measure the software quality is code coverage and complexity. Janzen and Saiedian (2008) examined whether TDD improves software design quality. Their work provides a basis for discussing design characteristics associated with test-first development. However, its findings should not be treated as universal evidence that TDD consistently improves every measure of code coverage or complexity. As captured in Figure 4, this study clearly reveals that in TDD, the code coverage improvement is not clearly witnessed however the nature of TDD work flow significantly influence the software design which ultimately simplifies the software code and hence reduces the overall code complexity in all dimensions.

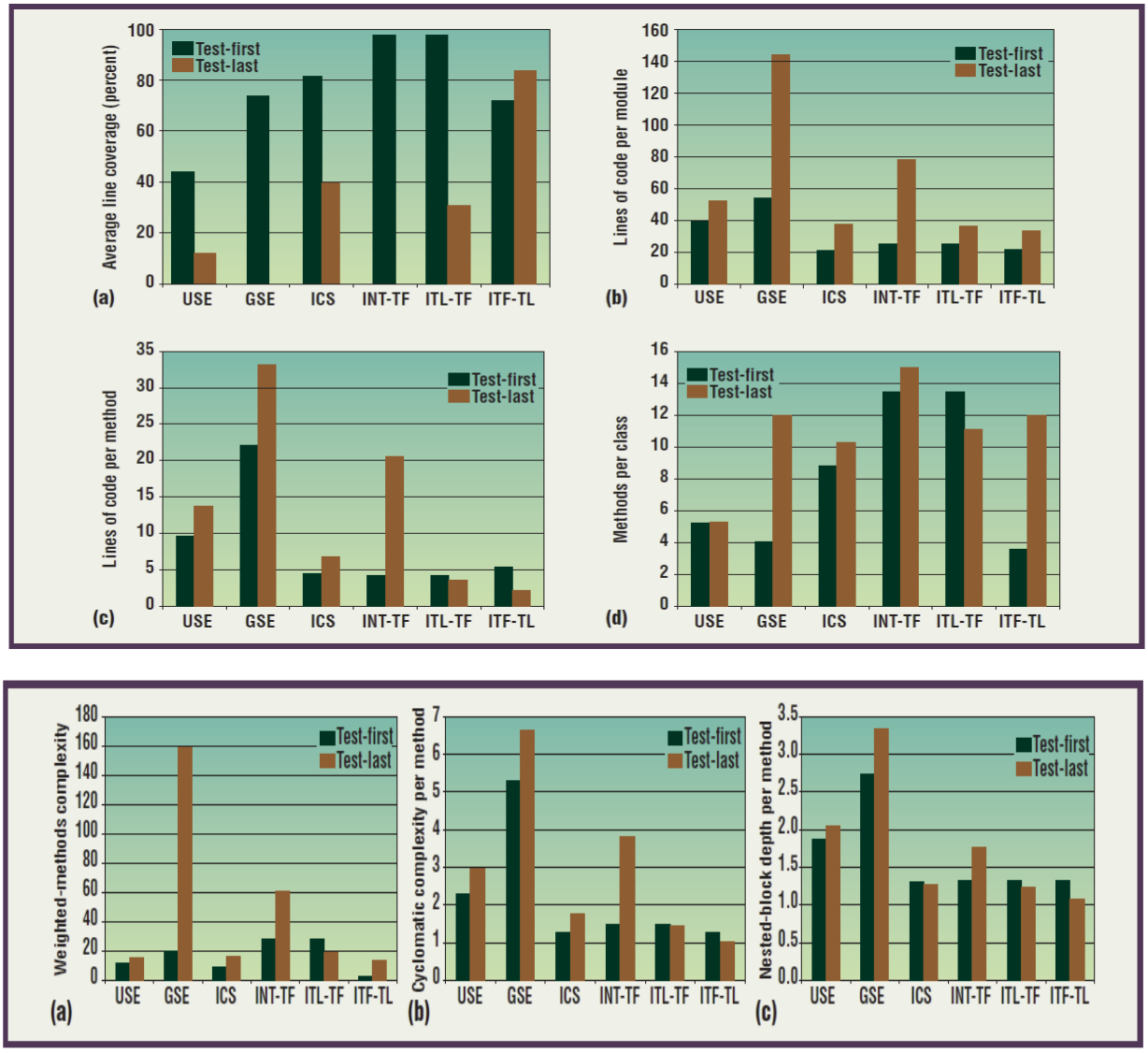
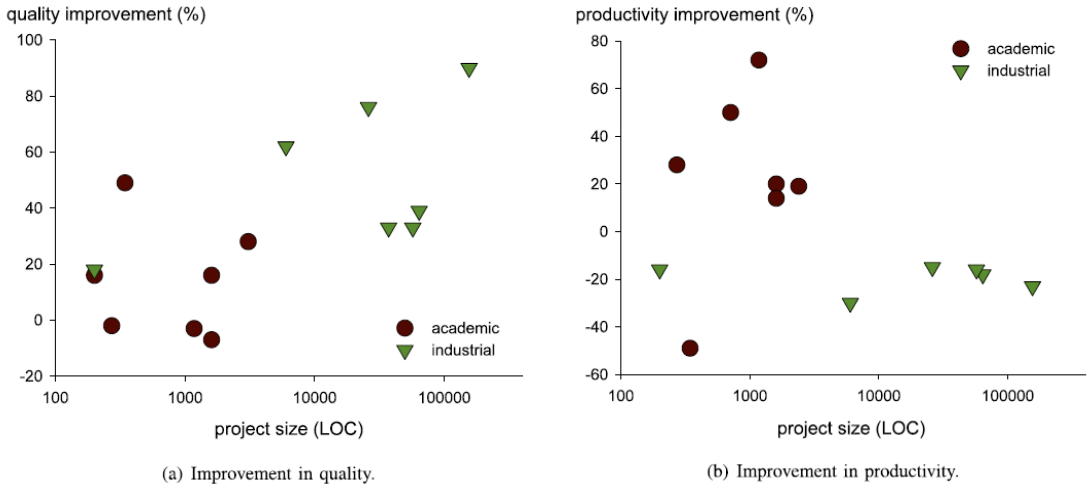


Figure 4. Impact of TDD on code coverage and complexity (adapted from Janzen & Saiedian, 2008)

4.1.4. Influence of Project Size

In the same study (Rafique & Mišić, 2013), it is also witnessed that the size of the project plays an important role in terms of quality and productivity in adopting TDD. As seen in Figure 5, as the software lines of code (LOC) increases the quality certainly improves in TDD however the productivity drops significantly.



(a) Improvement in quality. (b) Improvement in productivity.

Figure 5. Quality and productivity impact (adapted from Rafique & Mišić, 2013)

5. TDD in Embedded Systems

Test-Driven Development (TDD) can support embedded software development by encouraging small, testable changes and early defect detection. However, embedded systems introduce challenges that general-purpose software studies may not capture. These include hardware dependencies, real-time behaviour, limited memory, target-specific toolchains, and safety requirements. Effective embedded TDD therefore requires a testing strategy that combines host-based unit tests, target-based tests, hardware integration tests, and system-level validation. Cui (2024) and Dohmke (2008) provide relevant starting points for discussing TDD in embedded development. However, the benefits depend on the system architecture, available test infrastructure, and verification requirements.

5.1. Testing Frameworks for Embedded C and C++

Framework selection affects the practicality of embedded TDD. Unity is a unit-testing framework designed for C. CppUTest supports testing C and C++ software. These frameworks can help developers create repeatable tests for embedded application logic and hardware abstraction interfaces.

The choice of framework should reflect the programming language, compiler, target platform, and available build system. Teams should also consider test execution speed, debugging support, and integration with continuous integration pipelines. Framework adoption does not guarantee software quality. Test effectiveness still depends on meaningful test cases, adequate coverage, and additional target-level verification.

5.2. Safety Standards and Coding Requirements

Safety-critical embedded systems require verification beyond ordinary unit testing. Depending on the application, relevant standards may include ISO 26262 for road-vehicle functional safety, DO-178C for airborne software, and IEC 61508 for functional safety in electrical, electronic, and programmable electronic systems. MISRA guidelines also provide coding recommendations for safety-related embedded software.

TDD can support early defect detection and repeatable regression testing. However, adopting TDD does not establish compliance with these standards. Projects must follow their applicable lifecycle, verification, documentation, traceability, and assurance requirements. Tests should link to documented software requirements where required. Reviews, static analysis, integration testing, and system validation may also be needed.

5.3. Key Challenges in Large-Scale Embedded Systems

Developing large-scale embedded software systems had unique challenges, including resource constraints, real-time performance requirements, complex hardware-software integration, and security vulnerabilities, all while maintaining stability and safety. The top level classification of these challenges are "Complexity" , "Quality" & "Cost".

Embedded software must often satisfy timing deadlines and resource limits. Conventional unit tests can verify functional outputs but cannot independently establish real-time performance. Interrupt latency, scheduling behaviour, communication delays, and hardware contention require additional testing. Developers should therefore combine unit tests with timing measurements and target-based integration tests.

Memory limitations also affect embedded testing. Test frameworks, instrumentation, and diagnostic code can increase resource usage. Compiler options and processor architectures may further influence program behaviour. Tests should use representative build configurations wherever practical. Qualification should also consider stack usage, memory consumption, compiler warnings, and target-specific constraints. Passing unit tests alone does not demonstrate compliance with real-time requirements.

5.3.1. Complexity

Complex systems have always existed, however in recent decades there has been a profound increase in the level of complexity and industry must cope with the rapid increase in the rate at which complexity is accelerating (MarketsandMarkets, n.d.). Some of the examples include driverless cars; autonomous Urban Air Vehicles (UAVs); Lithography machines & quantum computing; 5G, the next generation of mobile connectivity technologies.

5.3.2. Quality

Embedded systems are widely used in safety-critical applications, where malfunctions can have serious repercussions, such as loss of life, property damage, or financial losses, therefore quality is always a crucial part in any embedded system. Strict quality assurance and control processes are required to ensure dependability, safety, compliance, and performance.

5.3.3. Cost

The development cost and the maintenance costs of these sophisticated embedded systems are generally significant. Therefore, any defect reported at the latter stage of product life cycle could be very expensive. Depending on the severity of the issue and the stage at which it is found, repairing bugs in any large and complex embedded software systems (a good example could be, lithography machines) is very expensive. With real-time embedded software managing incredibly precise motions, optics, and exposure parameters, lithography machines are extremely complex devices. In a typical semiconductor fabs, or fabrication plants, unscheduled downtime can cost up to $1 million every day. Minor UI or parameter changes could be resolved with a $10,000–$50,000 software patch, however a critical Real-time Control Bugs require in-depth analysis, impact machine accuracy, and can cost anywhere from $100,000 to $1 million or more. Average machine downtime cost is ~$30 per second (~$2.5 Million) for any unscheduled downtime on a high-end ASML lithography machine.

5.4. Mitigation by TDD

5.4.1. Complexity

TDD encourages developers to divide software requirements into small, testable behaviours. In embedded systems, this approach can improve the separation between application logic and hardware-specific code. Hardware abstraction and test doubles can make individual components easier to test. However, the extent of design improvement depends on

implementation quality and system architecture. Existing general-purpose TDD findings do not establish that TDD always outperforms test-last development in embedded projects.

5.4.2. Quality

Software quality is particularly important in large and complex embedded systems. The selected literature suggests that TDD can support software quality in some development contexts. However, the available evidence does not justify claiming that TDD consistently matches or outperforms test-last development in every experimental study or embedded application.

5.4.3. Cost

TDD may increase initial effort when teams create test infrastructure, hardware abstractions, and automated test suites. However, reusable tests can support regression testing and reduce repeated manual checks. The overall cost depends on system complexity, test execution time, debugging effort, and maintenance needs. AI tools may assist with test drafting and repetitive coding, but their outputs require review and verification. Their cost-saving effects in complex embedded systems require further empirical evidence.

5.4.4. Host-Based and Target-Based Testing

Embedded TDD can use both host-based and target-based testing. Host-based tests execute on a development computer rather than the target device. They support rapid feedback and simplify debugging of application logic. Target-based tests execute on the intended processor or embedded platform. They reveal issues related to hardware, compilers, memory layout, and target-specific behaviour. Host-based tests cannot fully reproduce the target environment. Differences in processor architecture, compiler behaviour, interrupts, and peripheral access can affect results. Therefore, passing host-based tests does not establish complete target compatibility. A practical strategy uses host-based tests during short development cycles. It then uses target-based and integration tests to verify hardware-dependent behaviour.

6. Influence of Artificial Intelligence in TDD

One of the major challenges in TDD is that it requires iterative coding (both production and test) which reduces productivity and hence increase the software development cost. In the age of generative AI (Gen-AI) many research is ongoing to automize TDD by embracing GenAI to create an auto generated code with minimal supervision on human developers. In a exploratory experiment with Chat GPT, discussed in research paper by Mock et al. (2025), it is found that Gen-AI can be efficiently used in TDD with the minimal involvement of developers (as shown in Figure 6).

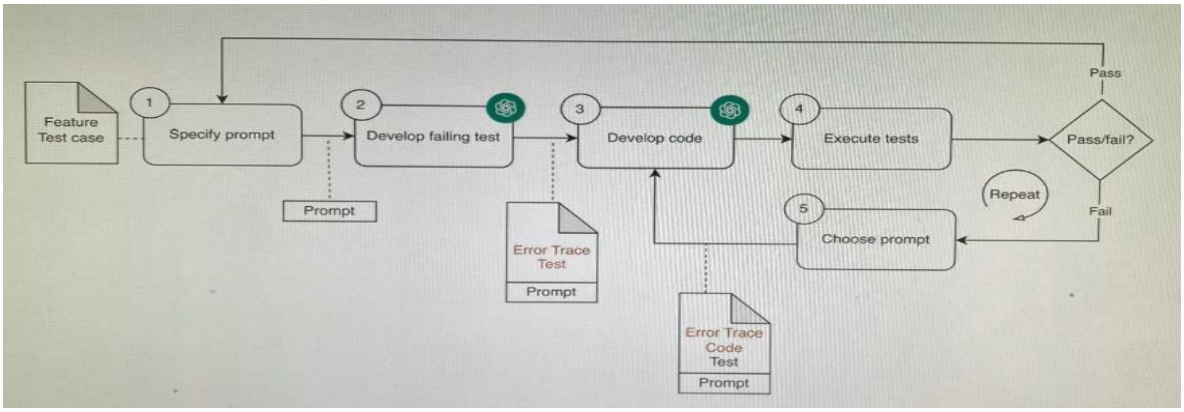


Figure 6. ChatGPT in TDD

The study of program synthesis has been ongoing for many years, and more recent methods have attempted to use large language models to aid in code generation. The research paper by Piya and Sullivan (2024) explores the concept of LLM4TDD, where Large Language Models are guided to generate code iteratively using a test-driven development methodology. Researcher conducted an empirical evaluation using ChatGPT and coding problems from LeetCode to investigate the impact of different test, prompt and problem attributes on the efficacy of LLM4TDD.

The results of this study led to several best practices for LLM4TDD, such as sanitizing method names and reducing ambiguity in test with careful consideration for integers. A lot of research is in progress to explore more opportunities in using AI to reducing the software development cost which is more significant in iterative development process in TDD. In another demo paper (Bala & Kumar, 2025) on Gen-AI in TDD is experimented in smaller embedded system projects with a PyCharm plugin to generate the production code. In all these experiments it was found that the Gen-AI plugin could successfully implemented the chunk of functionalities which eased the developer's job (of repetitive coding) and to focus more on re-factoring the generated code to improve the code quality. More progressive study is ongoing to improve the plugin to handle more complex requirements.

6.1. Limitations and Future Research

This paper presents a narrative overview rather than a systematic review. Much of the discussed empirical evidence concerns general-purpose software development. Therefore, its findings cannot automatically be generalised to embedded systems. Embedded TDD requires further investigation across host-based testing, target-based testing, hardware abstraction, real-time behaviour, and resource-constrained environments.

Future studies should evaluate TDD using representative embedded projects and documented experimental methods. They should measure defect detection, development effort, regression-test effectiveness, memory use, and timing performance. Studies should also examine how TDD fits within applicable safety-assurance processes. Comparisons should clearly describe the target hardware, toolchain, test environment, project size, and development approach. Such evidence would help establish where TDD provides practical benefits and where additional testing techniques remain necessary.

7. Conclusion

The main objective of this exploratory research work is to raise awareness of Test-Driven Development (TDD) as a design alternative approach and to assist professionals engaged in large and complex embedded software systems development in taking appropriate decisions before considering TDD over traditional water fall approach. This overview discusses selected research on TDD and its potential relevance to large-scale, complex embedded software development.

The reviewed publications highlight possible benefits for software design and quality, alongside uncertainty about productivity outcomes. These findings provide context for engineering decisions but do not constitute an exhaustive assessment of the literature. It is also concluded that TDD helps enhancing the software design simplicity, which ultimately influences the software quality however reduces the productivity. The impact of TDD, on productivity (both in development and maintenance, which is a key driving factor for industrial projects) remains a constant challenge.

To improve the efficiency (helps reducing the cost), modern AI tools/applications can play a significant role in future, where many researches are ongoing and still in preliminary stage. Therefore, for any large and complex embedded system development, TDD can play a

significant role in coming future, however more research and exploration needs to be done in future. This exploratory work may pave the way to conduct studies that produce more converging results to explore the potential benefits of TDD to overcome the challenges faced in large and complex embedded system development.

References:

Andrea, J. (2009). Test-driven development: Driving new standards of beauty. In A. Goucher & T. Riley (Eds.), *Beautiful testing: Leading professionals reveal how they improve software* (pp. 181–194). O'Reilly Media.

Astels, D. (2003). *Test driven development: A practical guide*. Prentice Hall.

Bala, A. K., & Kumar, A. (2025). Research progress knowledge mapping on transit oriented development (TOD) typology. *Engineering, Technology & Applied Science Research, 15*(2), 21847–21853. https://etasr.com/index.php/ETASR/article/view/10097

Beck, K. (1999). *Extreme programming explained: Embrace change*. Addison-Wesley.

Beck, K. (2003). *Test-driven development: By example*. Addison-Wesley.

Consortium for Information and Software Quality. (2023). *The cost of poor software quality in the US: A 2022 report*.

Cui, J. (2024). *A comparative study on the impact of test-driven development (TDD) and behavior-driven development (BDD) on enterprise software delivery effectiveness* (arXiv:2411.04141). arXiv. https://arxiv.org/abs/2411.04141

Dohmke, T. (2008). *Test-driven development of embedded control systems: Application in an automotive collision prevention system* [Doctoral dissertation, University of Glasgow]. https://doi.org/10.5525/gla.thesis.239

Dojo Five. (n.d.). *Unit testing for embedded software development*. https://dojofive.com/blog/unit-testing-for-embedded-software-development/

Erdogmus, H., Morisio, M., & Torchiano, M. (2005). On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering, 31*(3), 226–237.

Janzen, D. S., & Saiedian, H. (2008). Does test-driven development really improve software design quality? *IEEE Software, 25*(2), 77–84. https://doi.org/10.1109/MS.2008.34

MarketsandMarkets. (n.d.). *Embedded system market*. Retrieved April 1, 2023, from https://www.marketsandmarkets.com/Market-Reports/embedded-system-market-98154672.html

Matcha, S., & Singh, S. P. (2025). *Test-driven development (TDD) and its impact on software maintainability: An empirical study*. ResearchGate. https://www.researchgate.net/publication/393468386

Mock, M., Melegati, J., & Russo, B. (2025). Generative AI for test driven development: Preliminary results. In *Agile processes in software engineering and extreme programming – Workshops (XP 2024)* (Lecture Notes in Business Information Processing, Vol. 524, pp. 24–32). Springer. https://doi.org/10.1007/978-3-031-72781-8_3

Piya, S., & Sullivan, A. (2024). LLM4TDD: Best practices for test driven development using large language models. In *Proceedings of the 2024 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)* (pp. 14–21). ACM. https://doi.org/10.1145/3643795.3648382

Rafique, Y., & Mišić, V. B. (2013). The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering, 39*(6), 835–856. https://doi.org/10.1109/TSE.2012.28

Ranjan, K., Chandra, R., & Lulla, K. (2025). Enhancing reliability in complex embedded systems software: The critical role of unit testing in the age of AI and machine learning. *International Journal of Computational and Experimental Science and Engineering, 11*(3), 4170–4175. https://doi.org/10.22399/ijcesen.2633