Test Driven Development In The AI Era

Ivan Assis
Departamento de Ciência da Computação, Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
ivan.assis@gmail.com

Marco Túlio Valente
Departamento de Ciência da Computação, Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
mtov@dcc.ufmg.br

Abstract

Large language models accelerate software development, but the code and tests they generate often lack the discipline test-driven development (TDD) is meant to enforce. This paper proposes AI-TDD, a methodology that operationalizes the Red-Green-Refactor cycle through three role-based prompt personas — an Architect, a Test Writer, and an Implementer — and evaluates it as a comparative case study against Code-First, a conventional AI-assisted development baseline, on a custom-built e-commerce pricing engine with documented state interdependence across ten cross-mechanism business rules. Considered in isolation, the primary metrics do not show an unconditional advantage for AI-TDD: Code-First produced lower cyclomatic complexity and higher Mutation Score and test coverage. AI-TDD, in turn, sustained a 100% Fail-to-Pass rate across every genuinely attempted cycle, correctly generalized business rules ahead of schedule, and produced code its own validator found easier to follow one criterion at a time. The observed quality gap traces to two specific, addressable mechanisms — architectural conformance and under-tested generalized behavior — rather than an inherent limitation of AI-driven test-first discipline. This work formalizes the AI-TDD methodology as reusable personas, contributes an empirical evaluation collecting Mutation Score on a high-complexity business-rule system, and characterizes the mechanisms behind a comparative quality gap rather than offering a simple verdict on which approach is better.

Keywords

Test-Driven Development, Large Language Models, AI-Assisted Software Development, Mutation Testing, Case Study

1 Introduction

The rise of Large Language Models (LLMs) is reshaping software development practices. Code generation tools powered by these models promise substantial productivity gains by automating implementation tasks that traditionally required significant developer effort — a controlled experiment with GitHub Copilot found that developers with AI assistance completed tasks 55.8% faster than those without it [16]. However, this acceleration introduces a critical quality risk: LLM-generated code frequently suffers from logical inconsistencies, hallucinated dependencies, and inadequate test coverage [19, 20]. In an empirical evaluation of ChatGPT for unit test generation, Yuan et al. found that 57.9% of generated tests failed to compile and 17.3% failed at runtime, revealing that AI-generated artifacts may create an illusion of quality that does not hold in practice. This risk compounds in large-scale systems, where the absence of a robust test suite allows defects to propagate silently across interdependent components, significantly increasing the cost of maintenance and correction over time [15].

Test-Driven Development (TDD), formalized by Beck [3], offers a disciplined counterpoint to this quality challenge. By enforcing a strict Red-Green-Refactor cycle — where a failing test must precede any implementation — TDD structurally prevents the accumulation of untested code and promotes modular, low-complexity designs. Empirical evidence supports these benefits: studies show that TDD can reduce pre-release defect density by 40% to 90% in industrial settings [15]. Crucially, research by Fucci et al. suggests that the primary driver of these benefits is not the test-first ordering itself, but the fine-grained granularity of development cycles it enforces — the disciplined practice of small, incremental "baby steps" that Beck himself prescribes [3] — working in small, uniform increments consistently yields higher quality outcomes [7]. Despite this, TDD has not achieved widespread industrial adoption. Causevic et al. identify several limiting factors that systematically prevent its use in practice: increased development time, the steep learning curve required to write effective tests, and the difficulty of maintaining strict cycle discipline under deadline pressure [5]. In essence, TDD is mechanically sound but practically constrained by the human cost of executing it consistently.

The emergence of LLMs suggests a potential solution to these human bottlenecks. If AI prompt personas can reliably execute the Red-Green-Refactor cycle, the primary barriers to TDD adoption — time, skill, and discipline — could be substantially mitigated. Recent work supports this hypothesis: Mathews and Nagappan demonstrate that providing tests as input to LLMs acts as a formal specification, reducing ambiguity and improving code generation accuracy [12]. The Red phase of the TDD cycle requires by definition that a test must fail before any implementation exists [3]. When an AI system executes this cycle, the *Fail-to-Pass* property — a test must fail against the empty codebase and pass after implementation — becomes the direct criterion for validating that the model follows genuine test-first discipline rather than generating tests post hoc. Empirical benchmarks confirm that current LLMs sustain this property in fewer than one in four cases, and that when they do, the resulting tests exhibit substantially higher coverage [1]. A further limitation of existing studies concerns evaluation depth: most rely on code coverage as the primary quality metric, which measures execution completeness but not fault detection capability. Madeyski demonstrates that Mutation Score is a superior indicator of test suite robustness, as it directly evaluates whether tests can detect injected defects [11]. Taken together, these gaps point to an unaddressed research question: whether AI-TDD delivers measurable quality benefits — in terms of cyclomatic complexity, *Fail-to-Pass* rate, and Mutation Score — when applied to a system with high business-rule complexity and strong state interdependence.

To address this research question, this paper proposes and empirically evaluates AI-TDD, a methodology that adapts the Red-Green-Refactor cycle through specialized prompt personas — role-based markdown templates that define the role, constraints, and expected output format for each development phase. To validate AI-TDD under conditions of high complexity, we conduct a controlled experiment using a custom-designed e-commerce pricing engine as the object of study — a system chosen for its high native cyclomatic complexity, strong state interdependence, domain familiarity to practitioners and reviewers, and its absence from public code repositories, which mitigates the risk of data leakage in LLM-based experiments [13]. The AI-TDD approach is evaluated against a conventional AI-assisted development baseline (Code-First), using cyclomatic complexity, *Fail-to-Pass* rate, and Mutation Score as primary evaluation metrics.

This work makes the following contributions:

- A formalization of the AI-TDD methodology with role-based personas;
- An empirical evaluation of AI-TDD on a high-complexity business-rule system, collecting Mutation Score as a primary quality metric for test suite robustness assessment;
- A comparative analysis of AI-TDD versus Code-First development across quality and efficiency dimensions, under controlled conditions designed to isolate the effects of disciplined test-first practice.

2 Proposed Solution

This work proposes AI-TDD, a methodology that operationalizes the Red-Green-Refactor cycle through role-based prompt personas. We evaluate AI-TDD through a comparative case study against a conventional AI-assisted baseline, Code-First, applying both to a single object of study under controlled conditions. Runeson and Höst characterize case studies as suited to settings with many more variables of interest than data points, relying on multiple, triangulated sources of evidence to build a deeper understanding of the phenomenon under study rather than the causal, statistically generalizable results an experiment would target [18] — a description that matches this evaluation directly: a single execution per condition, but a system with dozens of interacting specification units, and both quantitative metrics and qualitative analysis collected throughout. The remainder of this section describes the AI-TDD methodology (Section 3.1), the Code-First baseline (Section 3.2), the object of study (Section 3.3), the specification protocol (Section 3.4), the experimental controls (Section 3.5), and the operational definition of each evaluation metric (Section 3.6).

2.1 AI-TDD Methodology

AI-TDD operationalizes the Red-Green-Refactor cycle through three role-based personas: an Architect, a Test Writer, and an Implementer. The Architect runs once, producing a shared architectural skeleton — a fixed decomposition into modules, complete data models, and stubbed function signatures — that both the Test Writer and the Implementer use as a stable reference throughout the remainder of the cycle. For each acceptance criterion, the Test Writer and the Implementer then alternate: the Test Writer drafts a failing test that encodes the criterion, initiating a Fail-to-Pass cycle,

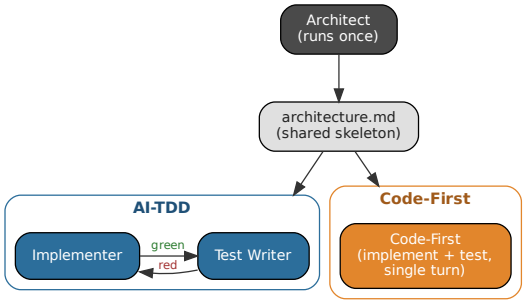

Figure 1: The AI-TDD cycle — Architect, Test Writer, and Implementer — contrasted with the Code-First baseline.

and the Implementer modifies the code until that test, and every previously passing test, succeeds. Each persona is operationalized as a structured prompt template — a markdown file specifying its role, constraints, and expected output format — following the open Agent Skills specification [2]; we refer to each such template as a skill.

This per-criterion granularity is a deliberate design choice, not an incidental one. Beck's original prescription for sustaining TDD's discipline is to proceed in small, incremental "baby steps" [3], and Fucci et al. provide empirical support for this prescription, showing that the granularity of development cycles, not the test-first ordering itself, is what drives TDD's quality benefits [7]. Calais and Franzini further connect this same granularity to developer experience, arguing that the short, well-defined feedback loops of the Red-Green-Refactor cycle create the preconditions associated with flow state during development [4]. This granularity also keeps a human reviewer in the loop: because each cycle addresses one narrowly scoped criterion, the developer supervising AI-TDD can follow how a specific behavior was implemented without reconstructing intent from a large, already-complete change — the same comprehension benefit long reported for human-driven TDD [6, 14], here extended to a setting where the developer supervises, rather than writes, each step.

Unlike Beck's original three-phase cycle, AI-TDD does not assign Refactor to a dedicated persona. Refactoring is instead folded into the Implementer as a conditional, narrowly scoped step: it is only triggered by obvious duplication or unclear naming that can be resolved without altering any test's behavior, excluding broader structural redesign. Figure 1 illustrates the resulting cycle, contrasting it with the Code-First baseline described next.

2.2 Code-First Baseline

Code-First represents conventional AI-assisted development: a single skill receives the full specification for one unit and, in a single turn, produces both the implementation and its accompanying tests, with tests written after the implementation rather than before. Like AI-TDD, Code-First starts from the same shared architectural skeleton (Figure 1), so any difference observed between the two

conditions can be attributed to the development process itself, not to a different starting point. Code-First was chosen as the baseline because it reflects how AI coding assistants are typically used in practice today, without an explicit test-first protocol — a realistic point of comparison for AI-TDD's promised benefits, not a deliberately weakened one.

2.3 Object of Study

The object of study is a deterministic e-commerce pricing engine that computes a final order total from a customer's cart, loyalty tier, applied coupons, redeemed points, payment method, and delivery context, through a fixed seven-step pipeline (subtotal computation, per-item promotions, coupon application, loyalty and points, shipping, payment-method adjustment, and finalization). Six coupon types and four loyalty tiers interact through the ten explicit cross-mechanism rules listed in Table 1, producing the kind of state interdependence and edge-case density that a single, isolated function cannot reproduce.

Table 1: Cross-mechanism interactions in the pricing pipeline.

Trigger	Affects	Effect
Diamond-tier customer	Shipping	Automatically free
Diamond-tier customer	Coupon stacking	Percentage and fixed-value coupons cannot be applied together
Gold or Diamond tier	VIP coupon eligibility	Unlocks eligibility otherwise blocked
Diamond tier with VIP coupon applied	VIP discount	Multiplied by 1.5
Points redeemed in the order	Loyalty multiplier on points earned	Collapses to 1.0
Customer has an open order	First-purchase coupon	Rejected
Customer has an open order	Points earned	Forced to zero
Percentage and fixed-value coupons both applied	Diamond customer	Mutually exclusive (fixed-value rejected)
Subtotal above the free-shipping threshold	Shipping	Becomes free; Free-Shipping coupon rejected as redundant
Final total within a small positive range	Final total	Clamped to a minimum; excess discount discarded

Two further design choices keep this object of study valid for what this evaluation measures. First, the pipeline is deterministic and has no external dependencies — no database, no network calls, no web framework — so every function is a pure transformation of its inputs, testable without mocks or fixtures unrelated to the business rules themselves; this matters specifically for Mutation Score, since mutations land exclusively in business logic, and a surviving mutant can be attributed to a gap in test coverage of business rules rather than to framework boilerplate the tests were never meant to exercise. Second, the engine was custom-designed for this study and is absent from public code repositories, mitigating the risk that the language models under evaluation had memorized its implementation during training [13].

2.4 Specification Protocol

This evaluation organizes work into specification units: structured specifications with acceptance criteria, each composed of a short capability title, a reference to the formal domain specification it derives from, a natural-language description of the expected behavior, a set of testable acceptance criteria — including concrete input-output examples for boundary conditions — and an explicit statement of what is out of scope. The specification precedes development and serves as a fixed reference throughout the cycle; it is not regenerated or treated as a source of truth after implementation begins. The 27 specification units used in this evaluation are delivered incrementally, one at a time, in the same fixed order for both AI-TDD and Code-First.

2.5 Experimental Controls

Beyond the shared architectural skeleton and specification protocol described above, four further controls were held constant across both conditions. First, the same language model and configuration were used throughout: Claude Sonnet 4.6 at temperature zero, fixed for the entire evaluation and never varied between conditions. Second, function, class, and method names were sanitized in every prompt, following Piya and Sullivan's recommendation that descriptive names bias an LLM's generated behavior toward what the name suggests rather than what the specification states [17]. Third, AI-TDD and Code-First were each developed in an isolated repository, with no shared history or cross-contamination between conditions. Fourth, a human reviewer supervised both conditions at defined stopping points — after each Test Writer/Implementer cycle and at the end of each specification unit in AI-TDD; after the production code and after the test suite in Code-First — and could request changes from the skill whenever the generated content did not conform to the specification. This oversight role was symmetric across both conditions: the reviewer's authority to request a correction, not just to approve completion, was the same in both arms.

Session boundaries differed between conditions to match each one's structure. AI-TDD opened two persistent sessions per specification unit — one for the Test Writer, one for the Implementer — both kept open across every acceptance criterion in that unit and closed only when moving to the next. Code-First opened a single session per specification unit, since one turn produces both the implementation and its tests.

2.6 Evaluation Metrics

This evaluation reports the three primary metrics named in the research question — cyclomatic complexity, Fail-to-Pass rate, and Mutation Score — since these are what the research question asks

about. Cyclomatic complexity is reported as the average and the maximum value per function in the source tree. Secondary metrics collected alongside them — line coverage and branch coverage via code execution tracing, and development time via session timestamps — are introduced directly alongside the results they support in Section 4, rather than defined here.

Fail-to-Pass rate operationalizes the property introduced in Section 1: a test must fail against the codebase before any implementation addressing it exists, and pass afterward, without causing any previously passing test to fail. For each acceptance criterion, the new test is run once before the Implementer's change and once after; a cycle counts toward the Fail-to-Pass rate only if both conditions hold. This property matters because it is the only direct evidence that a persona followed genuine test-first discipline rather than writing a test to match code that already existed after the fact. Ahmed et al. raise this same concern at the level of LLMs in general, benchmarking how often current models sustain Fail-to-Pass when generating tests; they find fewer than one in four cases succeed, which is precisely the gap AI-TDD's protocol is designed to close by construction rather than leave to chance [1].

Mutation Score is computed with mutmut as the ratio of killed mutants to the total number of non-equivalent mutants, $killed/(killed + survived)$ [10], where the denominator excludes mutants that cannot be conclusively classified as killed or survived: equivalent mutants, which by definition can never be killed, and mutants that time out during execution, whose outcome cannot be observed. This evaluation adopts Mutation Score over coverage-based metrics for two reasons. First, Madeyski directly measured the impact of test-first programming on this indicator, finding that test-first development produced test suites with higher Mutation Score than test-last development under otherwise identical conditions [11] — evidence from the test-first literature itself that the metric is sensitive to the discipline this evaluation investigates, not a generic substitute for coverage. Second, a test suite can reach high line or branch coverage while still failing to assert the behavior it exercises; Mutation Score requires a test to actually detect an injected fault to count, a stricter measure of whether tests verify behavior rather than merely execute it.

3 Evaluation

This section evaluates AI-TDD against Code-First on the same e-commerce pricing engine, one execution of each condition across all 27 specification units in the same fixed order, under the protocol and controls described in Section 2. Section 3.1 reports the resulting measurements, Section 3.2 discusses the mechanisms behind them, and Section 3.3 closes with the threats to validity inherent to this single-execution design.

3.1 Results

Table 2 reports the three primary metrics defined in Section 2.6, two structural counts included for context, and the secondary metrics collected alongside them. AI-TDD sustained a 100% Fail-to-Pass Rate, the one metric that does not apply to Code-First by design; Code-First scored higher on every other metric in the table.

Table 2: AI-TDD vs. Code-First.

Metric	AI-TDD	Code-First
Cyclomatic Complexity (avg)	2.65	2.49
Cyclomatic Complexity (max)	35	15
Fail-to-Pass Rate	100% (24/24)	N/A
Mutation Score	60.07%	73.84%
Functions in `src/`	26	39
Number of tests	32	116
Line Coverage	97.49%	99.01%
Branch Coverage	87.93%	96.25%
Total time	214.9 min	110.95 min
Mean time / unit	7.96 min	4.11 min

3.2 Discussion

AI-TDD's higher maximum complexity (35 vs. 15) and lower function count (26 vs. 39) are consistent with a difference in how the two conditions followed the shared architectural skeleton described in Section 2.1. Code-First's source tree mirrors the module boundaries the skeleton prescribes; AI-TDD's does not — the functions the skeleton assigns to a dedicated pipeline-steps module were instead folded into the function that orchestrates the entire pipeline. A plausible explanation is that AI-TDD's per-criterion focus gave no single cycle a reason to restructure code that already passed its own test, while Code-First's single-turn view of the whole unit preserved the prescribed decomposition by default. This explanation was not isolated by a controlled measurement in this study. This points to an addressable cause, not an inherent limitation of the discipline: a periodic architectural-conformance check could intervene before complexity accumulates unchecked, though this study did not implement or evaluate such a mechanism.

Mutation Score and Branch Coverage require a different explanation, since mutation testing operates on individual lines rather than module boundaries: decomposing a function differently does not by itself change how many mutants survive. A portion of AI-TDD's later specification units were resolved by confirming behavior already generalized from earlier units, rather than by new test-driven development. This same constraint explains part of AI-TDD's substantially smaller test suite (32 tests against Code-First's 116, Table 2): confirmed-ahead behavior accumulates no dedicated test of its own. A second, more structural source compounds this: a test asserting that valid input does not raise an error typically passes even against an empty implementation, disqualifying it as a Fail-to-Pass candidate under this protocol. AI-TDD's validation tests are accordingly almost entirely error-triggering cases; Code-First, with no such constraint, tests the valid case about as often as the error case for the same rules. This pattern has two sides: in every such case, the generalized behavior was already consistent with the later unit's own acceptance criteria, meaning the Implementer's early generalization anticipated the broader rule correctly — the gap this leaves is in dedicated testing of that behavior, not in its

correctness. Branch Coverage is consistent with the testing side of this pattern: the gap between conditions is far larger for branches (87.93% vs. 96.25%, 8.3 points) than for lines (97.49% vs. 99.01%, 1.5 points), suggesting that code exercised incidentally by other units' tests, rather than by a test written with intent toward its own logic, is more likely to leave one side of a conditional unexercised. This is a plausible explanation for the Mutation Score gap as well, though this study does not trace individual surviving mutants back to the unit that introduced the line they occupy. The same diagnosis suggests a targeted fix: a mutation-hardening step applied specifically to confirmed-but-untested behavior, rather than broad additional testing, is a natural candidate to close this gap — also untested here.

The 100% Fail-to-Pass Rate reported in Table 2 holds only over cycles genuinely attempted: of the 27 specification units, 21 required at least one such cycle, while 6 were resolved entirely by confirming that the required behavior already existed, with no new test ever becoming a Fail-to-Pass candidate. This confirms that the protocol's pre-implementation check works as intended — it distinguishes genuine red-green cycles from cases where the required behavior already existed, instead of conflating the two and inflating the apparent success rate.

AI-TDD took roughly twice as long per unit as Code-First (7.96 vs. 4.11 minutes) despite the six units above resolving without new development. The discipline that sustains the Fail-to-Pass check has a time cost even when its answer is "no further test is needed" — consistent with increased development time being among the barriers Causevic et al. identify for human-driven TDD adoption [5], here persisting under AI execution as well.

Two further observations apply to both conditions, not just one. First, both repositories show at least one instance of implementation proceeding ahead of its corresponding specification unit, traceable to a field comment in the shared architectural skeleton that revealed a later business rule ahead of time — a leakage vector neither protocol was designed to prevent. Second, both conditions required human-directed correction of a previously passing test when a later, correct rule conflicted with an earlier test's incomplete assumption. Code-First's single-turn view of each unit did provide one capability AI-TDD's per-criterion design has no equivalent for: at least one ordering defect and one pre-existing test inconsistency were caught through what amounts to a holistic audit at the start of a turn, before any new code was written. This is a structural difference between the two approaches, not evidence that Code-First is the better methodology — AI-TDD's design simply has no comparable point in its cycle for this kind of audit to occur. Conversely, AI-TDD's granularity had a qualitative benefit the metrics above do not capture: as this study's validator, reviewing one small, isolated change at a time made AI-TDD's resulting code and tests easier to follow than Code-First's larger, single-turn changes, consistent with the comprehension benefit this granularity was designed to preserve (Section 2.1). This impression reflects a single validator's experience, not an independently measured comparison.

3.3 Threats to Validity

This evaluation's threats to validity follow the standard internal, construct, and external categories.

Internal validity. A single validator — the author — reviewed both conditions, including the classification of which specification units required a genuine Fail-to-Pass cycle (Section 3.2), including the field-comment leakage that affected both conditions symmetrically; an independent second reviewer could classify these differently.

Construct validity. Fail-to-Pass Rate measures only cycles that were attempted as red-green, not total behavioral coverage, as discussed in Section 3.2. Mutation Score carries a small execution variance (mutants whose outcome depends on machine timing), which the formula in Section 2.6 isolates structurally rather than averaging out across repeated runs.

External validity. As established in Section 2, this is a comparative case study with a single execution per condition (n=1) in a single domain, deliberately chosen over a statistically powered design to support a rich, contextualized account of this object of study [18]. A specification error identified in one unit during execution was corrected and is reported here for transparency, not concealed.

4 Related Work

The work closest to ours is LLM4TDD, where a single LLM implements code against tests a human writes one at a time, evaluated on isolated LeetCode problems and measured by success rate and prompt overhead [17]. This work differs on every one of those points: a dedicated persona writes the tests, not the human; the object of study is a single system of 27 interdependent specification units, not isolated problems with no shared state; and the metrics are cyclomatic complexity and Mutation Score, evaluated against an explicit Code-First baseline LLM4TDD does not include. LLM4TDD's reported 88.5% success rate is therefore evidence that an LLM can implement against a human-authored test suite for self-contained functions, not that the same discipline holds when the LLM also writes the tests and the underlying system has interdependent state.

Human-driven TDD's design-quality benefits are themselves contested. Nagappan et al., Fucci et al., and Maximilien and Williams report quality and defect-density gains from disciplined test-first practice [7, 14, 15], while Janzen and Saiedian attribute these gains specifically to smaller, less complex units [9] — the opposite of what this evaluation observes under AI execution, where the same discipline produced fewer, larger functions and a higher maximum complexity than Code-First. More broadly, Ghafari et al. characterize the empirical record on TDD's benefits as inconclusive [8], a description this evaluation's mixed results do not contradict. Section 3.2 attributes the complexity gap specifically to architectural conformance, not to a failure of test-first discipline itself.

5 Conclusion

Code-First produced lower cyclomatic complexity and higher Mutation Score and test coverage. AI-TDD, in turn, sustained perfect discipline on Beck's own Fail-to-Pass criterion [3] across every genuinely attempted cycle, correctly generalized rules ahead of schedule in every case this evaluation traced, and produced code its own validator found easier to follow, one criterion at a time. The gap on the other metrics traces to two specific, addressable mechanisms — architectural conformance and under-tested generalized behavior — not to an inherent limitation of AI-driven test-first discipline.

This work delivers on its three contributions. The AI-TDD methodology is formalized as three role-based personas, operationalized as reusable skills (Section 2.1). The empirical evaluation collected Mutation Score, alongside cyclomatic complexity and Fail-to-Pass rate, on a system with documented state interdependence across ten cross-mechanism rules (Section 2.3, Section 3.1). The comparative analysis, rather than yielding a simple verdict, characterizes two specific mechanisms behind the observed quality gap and identifies structural differences in self-correction and comprehension between the two approaches (Section 3.2).

These findings carry the limitations already discussed in Section 3.3: a single validator, a single domain, and no statistical power beyond one execution per condition. Future work should:

- repeat this comparison across multiple executions and models to support statistical inference;
- involve independent validators to corroborate the qualitative classifications made here by a single author;
- investigate a coarser granularity in which the Test Writer drafts the full set of tests for a specification unit upfront rather than one criterion at a time — trading some of the cycle's incremental discipline for a reduction in the per-unit overhead this evaluation observed (Section 3.2), particularly for criteria whose behavior the Implementer has already generalized;
- investigate a relaxed Fail-to-Pass criterion, or a dedicated post-cycle persona, that credits tests confirming valid or already-correct behavior even when such a test cannot itself fail beforehand — the structural gap this evaluation identifies in AI-TDD's validation tests;
- investigate whether AI-TDD's granular cycles sustain the flow-state benefits reported for human-driven TDD [4], a question this evaluation did not measure.

Artifact Availability

The complete domain specification, the 27 specification units, the AI-TDD and Code-First repositories, and the skill definitions for all three personas are available at an anonymized mirror: https://github.com/IvanAssis07/ai-tdd-vs-codefirst-artifact. The repository will be made public upon acceptance.

References

[1] Toufique Ahmed, Martin Hirzel, Rahul Pan, Avraham Shinnar, and Saurabh Sinha. 2024. TDD-Bench Verified: Can LLMs Generate Tests for Issues Before They Get Resolved? *arXiv preprint arXiv:2412.02883* (December 2024). https://arxiv.org/abs/2412.02883

[2] Anthropic. 2025. Agent Skills: A Standardized Way to Give AI Agents New Capabilities and Expertise. https://agentskills.io/ Open specification, originally developed by Anthropic.

[3] Kent Beck. 2003. *Test-Driven Development: By Example.* Addison-Wesley Professional, Boston, MA, USA.

[4] Pedro Calais and Lissa Franzini. 2023. Test-Driven Development Benefits Beyond Design Quality: Flow State and Developer Experience. In *Proceedings of the 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER 2023).* IEEE, Melbourne, Australia, 106–111. doi:10.1109/ICSE-NIER58687.2023.00025

[5] Adnan Causevic, Daniel Sundmark, and Sasikumar Punnekkat. 2011. Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST).* IEEE, Berlin, Germany, 337–346. doi:10.1109/ICST.2011.19

[6] Chetan Desai, David Janzen, and Kyle Savage. 2008. A Survey of Evidence for Test-Driven Development in Academia. *ACM SIGCSE Bulletin* 40, 2 (June 2008), 97–101. doi:10.1145/1383602.1383644

[7] Davide Fucci, Hakan Erdogmus, Burak Turhan, Markku Oivo, and Natalia Juristo. 2017. A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last? *IEEE Transactions on Software Engineering* 43, 7 (July 2017), 597–614. doi:10.1109/TSE.2016.2616877

[8] Mohammad Ghafari, Timm Gross, Davide Fucci, and Michael Felderer. 2020. Why Research on Test-Driven Development is Inconclusive?. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2020).* ACM, Bari, Italy, Article 25, 10 pages. doi:10.1145/3382494.3410687

[9] David S. Janzen and Hossein Saiedian. 2008. Does Test-Driven Development Really Improve Software Design Quality? *IEEE Software* 25, 2 (March 2008), 77–84. doi:10.1109/MS.2008.34

[10] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62

[11] Lech Madeyski. 2010. The Impact of Test-First Programming on Branch Coverage and Mutation Score Indicator of Unit Tests: An Experiment. *Information and Software Technology* 52, 2 (February 2010), 169–184. doi:10.1016/j.infsof.2009.08.007

[12] Noble Saji Mathews and Meiyappan Nagappan. 2024. Test-Driven Development and LLM-based Code Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024).* ACM, Sacramento, California, USA, 1583–1594. doi:10.1145/3691620.3695527

[13] Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Robert Ma, Maxime Voisin, Eva Gilsenan-McMahon, and Matthias Gallé. 2024. On Leakage of Code Generation Evaluation Datasets. In *Findings of the Association for Computational Linguistics: EMNLP 2024.* Association for Computational Linguistics, Miami, Florida, USA, 13215–13223. doi:10.18653/v1/2024.findings-emnlp.772

[14] E. Michael Maximilien and Laurie Williams. 2003. Assessing Test-Driven Development at IBM. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003).* IEEE, Washington, DC, USA, 564–569. doi:10.1109/ICSE.2003.1201238

[15] Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. 2008. Realizing Quality Improvement Through Test Driven Development: Results and Experiences of Four Industrial Teams. *Empirical Software Engineering* 13, 3 (June 2008), 289–302. doi:10.1007/s10664-008-9062-z

[16] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. *arXiv preprint arXiv:2302.06590* (February 2023). https://arxiv.org/abs/2302.06590

[17] Sanyogita Piya and Allison Sullivan. 2024. LLM4TDD: Best Practices for Test Driven Development Using Large Language Models. In *Proceedings of the 1st International Workshop on Large Language Models for Code (LLM4Code 2024).* ACM, Lisbon, Portugal, 14–21. doi:10.1145/3643795.3648382

[18] Per Runeson and Martin Höst. 2009. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering* 14, 2 (2009), 131–164. doi:10.1007/s10664-008-9102-8

[19] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. doi:10.1109/TSE.2023.3334955

[20] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 1703–1726. doi:10.1145/3660783