Engineering Prompt-Orchestrated LLM Workflows for Automated Test Case Generation in Agile Environments

Almeyda Alania Fredy Antonio, Barrientos Padilla Alfredo, Siancas Garay Ronald Gustavo
Faculty of Engineering-School of Software Engineering, Peruvian University of Applied Sciences (UPC), Lima, Peru

***Abstract*—Manual test case generation for agile software development is a critical bottleneck that is costly, inconsistent, and error-prone. This study introduces a prompt-engineering and multi-level orchestration framework to automate this process. The proposed approach explicitly targets the automated generation of high-level acceptance test cases, addressing a gap in existing research that predominantly focuses on unit-level or reactive testing. The proposed tool, AI-Based Desktop Test Generator (AIDTG), employs a dual-LLM engine (Gemini 1.5 and GPT-4) to transform high-level functional descriptions from the Product Backlog into structured validation scenarios. Unlike prior LLM-based testing approaches, the framework integrates schema-aware prompt engineering and dual-model orchestration to ground the generation process in both functional intent and technical data constraints. The methodology is distinguished by its context-aware prompt engineering, which injects a frozen database schema to ground the models, and its ability to format outputs for the TestRigor BDD 2.0 platform. This schema-grounded and orchestrated workflow enables the systematic translation of informal User Stories into executable Behavior-Driven Development (BDD) acceptance tests, reducing ambiguity and improving semantic correctness. Experimental results on a real-world dataset of fifty User Stories show the framework reduces manual test design effort by eighty per cent, achieves a four point seven five (out of five) average quality rating from human experts, and produces BDD scripts with a ninety-one point nine per cent functional correctness pass rate. These results demonstrate that orchestrated, schema-aware Generative AI can operate as a reliable co-assistant for QA teams, improving efficiency while maintaining high standards of quality and executability.**

Keywords*—Software testing; test case generation; Large Language Models; Generative AI; prompt engineering; LLM orchestration; Behavior-Driven Development (BDD); agile methodology; acceptance testing; schema-aware prompting; Human-in-the-Loop; quality assurance automation

I. Introduction

The emergence of Generative Artificial Intelligence (AI) has initiated a profound paradigm shift across multiple sectors. Beyond its generalist use, its impact on labor productivity has been quantified; recent studies demonstrate that AI assistance can increase worker productivity by 15% on average, disproportionately benefiting less experienced workers [1]. The field of software engineering has been an early and significant beneficiary of this revolution. Large Language Models (LLMs) are being rapidly integrated into the software development lifecycle, demonstrating strong capabilities in code generation [28] and fundamentally changing developer workflows. This wave of transformation is now moving from code *creation* to code *validation*, showing considerable potential for optimizing long-standing challenges in Quality Assurance (QA) [8, 26].

Despite the velocity promised by agile methodologies and DevOps practices [25], quality assurance persists as the most significant critical bottleneck. The software industry overwhelmingly recognizes the manual creation of test cases as a process that is "tedious", "costly", and "susceptible to human errors" [6]. In modern CI/CD pipelines, where speed is paramount, this manual friction becomes untenable. Testing is estimated to consume a significant portion of the total development cost, becoming "less feasible" as software complexity increases [6]. Consequently, this crucial task is "often neglected", leading to the accumulation of technical debt and increasing the risk of defects being discovered only in late stages of development, or worse, by end-users.

To mitigate this friction, automation has been pursued in waves. First, traditional automation tools, such as EvoSuite [7], and pre-generative NLP approaches [14] offered assistance. However, these tools often generate tests that "lack readability and require manual intervention" [5] or are too brittle to handle the ambiguity of natural language requirements. The second wave, driven by the advent of LLMs, has shown greater promise. Systematic reviews confirm the growing use of AI for "Test Case Generation" and "Test Case Prioritization" [3]. In the academic sphere, frameworks have emerged exploring the generation of unit tests [5, 15, 16, 19], bug reproduction [4, 7, 27], and specialized API security tests [20, 22].

However, a critical analysis of these current LLM applications reveals a significant research gap. The vast majority of these approaches focus on low-level, code-centric tasks (unit tests) or *reactive* tasks (bug reproduction). While valuable, these solutions *do not address the primary agile workflow*: the proactive translation of high-level Product Backlog elements (User Stories) into business-facing acceptance tests. Existing tools often lack the *intelligent orchestration* [11] required to convert informal, non-technical requirements into high-level, context-aware QA artifacts that comply with Behavior-Driven Development (BDD) frameworks [13].

From an agile and DevOps perspective, this limitation is particularly critical, as acceptance testing represents the main validation mechanism for ensuring that delivered functionality aligns with business expectations before release.

This work directly addresses this gap. We introduce a system named the AI-Based Desktop Test Generator (AIDTG), a multi-level orchestration and prompt engineering framework designed to automate BDD acceptance test generation. The system moves beyond simple text-to-text translation by utilizing LLM APIs (such as GPT and Gemini) in concert with the crucial technical context of a frozen database *schema*. Through this advanced orchestration, AIDTG transforms high-level functional descriptions into semantically correct, data-aware validation scenarios compatible with TestRigor BDD 2.0 [23]. The framework acts as a co-assistant to improve the precision, coverage, and time efficiency of QA, generating execution-ready artifacts that bridge the gap between business requirements and technical validation.

Unlike prior academic approaches that either evaluate isolated prompt strategies or focus on code-level test generation, this work proposes an end-to-end, schema-aware orchestration framework explicitly tailored to acceptance testing in agile environments.

The core novelty of this research lies in three complementary contributions:

1) a dual-LLM orchestration strategy that separates logical test generation from strict syntactic formatting, 2) a schema-aware prompt engineering mechanism that grounds test generation in real database constraints, and 3) an integrated workflow that operationalizes the transformation of Product Backlog items into executable BDD acceptance tests.

Together, these contributions advance the state-of-the-art by moving beyond prompt-centric experimentation toward a reusable, extensible, and empirically validated orchestration framework for AI-assisted quality assurance.

The present study is organized as follows: Section II presents the study's theoretical foundations. Section III analyzes related work in the field. Section IV describes the AIDTG methodology and system architecture. Section V discusses the experimental results and evaluation. Finally, Section VI presents the conclusions and future lines of research.

II. Theoretical Foundations

This section establishes the theoretical groundwork underpinning the AI-Based Desktop Test Generator (AIDTG) framework.

Our system operates at the intersection of generative artificial intelligence, software testing theory, and agile quality assurance practices, integrating concepts from each domain to support automated acceptance test generation.

A. Artificial Intelligence and Large Language Models (LLMs)

Artificial Intelligence (AI) is a broad field of technology that enables computers and machines to simulate human learning, comprehension, and problem-solving [9]. Within this field, Large Language Models (LLMs) have emerged as a dominant force. LLMs are deep learning models, often built on a transformer architecture, that are trained on vast datasets to "work as giant statistical prediction machines" capable of understanding and generating natural, human-like language [10]. In software engineering, their application has rapidly moved from documentation to code generation, and most recently, to supporting complex quality assurance tasks [8]. Recent empirical studies indicate that LLMs can meaningfully augment human performance in software-related tasks, particularly when their outputs are constrained and guided by domain-specific context. However, unconstrained LLM outputs are prone to issues such as hallucination, inconsistency, and lack of executability, especially in tasks that require strict syntactic or semantic correctness, such as software testing.

B. Behavior-Driven Development (BDD)

The primary output of the AIDTG system is BDD-compatible scripts. Behavior-Driven Development (BDD) is an agile testing methodology designed to foster collaboration between developers, QA analysts, and business stakeholders [13]. It achieves this by defining application behavior in a structured, natural-language syntax known as Gherkin (e.g., Given-When-Then). This formal, high-level specification creates an unambiguous, "living" documentation that serves as both a requirement and an executable test script. BDD is particularly well-suited to agile environments, as it bridges the communication gap between technical and non-technical stakeholders while maintaining traceability between requirements and validation artifacts. Our work focuses on automating the translation of informal backlog items into this precise, high-level BDD format. By targeting acceptance tests rather than unit-level artifacts, the proposed framework aligns directly with the validation layer most closely associated with business value delivery.

C. Prompt Engineering and LLM Orchestration

A single, generic request to an LLM rarely yields a complex, executable artifact. Prompt Engineering is the methodology of structuring, refining, and optimizing the input (the prompt) to guide the LLM toward a more accurate and contextually relevant output [12]. LLM Orchestration extends this concept further; it is the process of managing, chaining, and coordinating the interactions of an LLM with external tools and data sources [11]. The AIDTG framework is an orchestration engine: it does not merely pass a user story to the LLM. It intelligently combines the user story (the intent) with the database schema (the technical context) via a multi-level prompt workflow to generate a valid, data-aware BDD script. This separation of responsibilities between prompt design and execution flow control reflects emerging best practices in LLM-based system engineering, where orchestration is treated as a first-class architectural concern rather than an implementation detail

D. Schema-Aware and Risk-based Testing

The most significant challenge in agile testing is ensuring test coverage for critical business logic [6]. A key aspect of our methodology is the use of a "frozen" database schema. This approach is informed by the principles of Risk-Based Testing (RBT), which uses risk assessments to "steer all phases of the test process to optimize testing efforts" [2]. By making the LLM "aware" of the data schema, AIDTG generates test cases that are not just syntactically correct BDD but are also semantically aware of the underlying data model. This schema-awareness allows the generation process to prioritize high-risk areas, such as data integrity constraints, entity relationships, and domain-specific validation rules, which are often overlooked in purely

text-driven test generation approaches. Consequently, the integration of schema-aware prompting with risk-based testing principles enables a more focused and meaningful acceptance testing strategy, improving both coverage quality and defect prevention potential.

III. Related Work

To situate the contribution of AIDTG, this section provides a detailed analysis of the current state-of-the-art in AI-driven test generation. The application of AI in software testing is a well-established field, with systematic reviews confirming its use for tasks like test case generation and prioritization [3]. The recent advent of Large Language Models (LLMs) has catalyzed a new wave of research [26], with industry surveys showing that QA professionals are already applying these models in practice [8]. However, the literature is highly fragmented, with most efforts diverging into distinct streams that fail to address the primary agile bottleneck of proactive, high-level, and context-aware acceptance testing.

To provide a clearer analytical perspective, existing work is discussed below according to its primary testing focus and level of abstraction, explicitly highlighting the limitations that motivate the proposed approach.

Historically, before the dominance of generative LLMs, research focused on traditional Natural Language Processing (NLP) and Model-Based Testing (MBT) to bridge the gap between requirements and tests. Works such as Lim et al. [14] proposed unified "boilerplate" approaches to extract test case information from requirements specifications using NLP. While structured, these methods are often brittle, struggling with the ambiguity of natural language. Concurrently, MBT, as reviewed by Ferrari et al. [29], exploits abstract models of software behavior to generate tests. Although effective in controlled settings, MBT approaches require the manual creation and maintenance of formal behavioral models, which introduces significant overhead and limits their adoption in fast-paced agile environments.

While powerful, this approach faces significant adoption barriers due to the high complexity and cost associated with creating and maintaining the formal models themselves. Other approaches utilized deep learning (pre-generative) to automate functional UI testing [30], but these remained focused on component-level validation rather than end-to-end business logic.

As a result, pre-LLM approaches generally fail to scale to acceptance-level testing scenarios where requirements are informal, rapidly evolving, and expressed in natural language.

The rise of LLMs [10] has shifted the research focus, with most efforts concentrating on code-level, specialized, or reactive testing tasks. A dominant trend is the generation of unit tests. Ouédraogo et al. [5] provided a large-scale evaluation comparing LLM-generated unit tests against traditional tools, noting LLMs produce more readable tests but often lack correctness. Subsequent research has attempted to mitigate these limitations by augmenting LLM-based unit test generation with additional techniques.

This stream includes enhancing unit test generation through mutation testing [15], using evolutionary algorithms to guide the LLM [16], or augmenting the process with assertion knowledge (A3Test) [19]. This entire body of work, while valuable for code-level validation, addresses a different problem than the high-level, business-facing acceptance testing that BDD targets. Specifically, unit test generation operates at the implementation layer and assumes access to source code, whereas acceptance testing operates at the requirement layer and must accommodate non-technical stakeholder input.

A second major research stream focuses on reactive testing—generating tests after a bug is found. Kang et al. [4] explored using LLMs as "few-shot testers" to reproduce known bugs, and Plein et al. [7] confirmed the feasibility of generating test cases from informal bug reports. These approaches demonstrate the effectiveness of LLMs in post-hoc validation scenarios, where the failure context is already known.

A recent extension of this, BRMINER [27], uses LLMs to extract relevant test inputs from bug reports to enrich existing test generation tools. This work is crucial for regression suites but does not address the primary agile bottleneck [6]: the proactive generation of tests from new functional requirements before they become bugs. Consequently, reactive LLM-based testing approaches improve defect reproduction but do not reduce the upfront effort associated with designing acceptance tests during sprint planning or backlog refinement.

A third stream applies LLMs to highly technical, specialized domains, such as developing self-improving frameworks for security testing of APIs using Karate DSL [20] [21] or optimizing REST API fuzzers [22]. These approaches, while advanced, are not aimed at validating the end-to-end business logic defined in agile user stories. Their domain specificity and reliance on technical testing artifacts limit their applicability to general-purpose acceptance testing workflows.

The most relevant, yet least explored, area is the application of LLMs to high-level acceptance testing, such as BDD. The work by Karpurapu et al. [13] is one of the few academic studies that directly addresses LLMs for BDD automation. However, their research is an evaluation of different prompt engineering techniques (zero-shot vs. few-shot) to formulate BDD tests. While informative, this work does not propose a complete, end-to-end orchestration framework that integrates external technical context to ensure the semantic and technical validity of the generated tests.

It does not propose a complete, end-to-end orchestration framework [11] that integrates external technical context (like a database schema) to ensure the generated tests are not just syntactically correct but semantically and technically viable. This limitation is particularly significant in real-world agile projects, where acceptance tests must align with underlying data models and business constraints.

This industry need is validated by the commercial success of tools like TestRigor [23], which positions itself as a "Generative AI-based Test Automation Tool" that allows teams to write tests in plain English [24]. Although such tools demonstrate the practical viability of LLM-driven acceptance testing, they are proprietary systems that do not expose their orchestration

strategies or prompt engineering mechanisms for academic analysis or replication.

This commercial validation proves the desirability of the approach, but it is a proprietary, closed-box solution, not an open academic framework.

Our comprehensive review thus reveals a clear and significant research gap. The literature is heavily skewed toward code-level [5, 16, 19], reactive [4, 7, 27], or niche [20, 22] test generation. The few works that touch upon high-level BDD testing [13] stop at prompt evaluation and do not provide an integrated framework. In particular, existing approaches lack systematic LLM orchestration mechanisms capable of combining requirement-level intent with technical system context.

A gap persists for a system that is: 1) Proactive, generating tests from Product Backlog items; 2) High-Level, focusing on BDD acceptance tests; and 3) Context-Aware, using LLM Orchestration [11] to integrate external technical artifacts.

This work addresses this precise lacuna. AIDTG is not a unit test generator or a bug-reproduction tool; it is an orchestration framework explicitly designed to automate the translation of high-level agile requirements into data-aware BDD acceptance tests [13]. By grounding generation in both functional requirements and database schemas, the proposed approach advances beyond prior prompt-centric solutions and directly targets the primary quality assurance bottleneck in modern agile and DevOps processes [25].

IV. Methodology

The methodology of this work consists of the design, implementation, and evaluation of an end-to-end LLM orchestration framework, named the AI-Based Desktop Test Generator (AIDTG). This system is architected to manage the full lifecycle of agile test case generation, from project creation to optional execution, acting as an intelligent co-assistant for QA teams [8].

The methodological approach follows an empirical software engineering paradigm, combining system design with experimental validation on real-world project data.

The system is designed as a decoupled, API-first application, comprising a Python-based backend for orchestration and a separate desktop frontend for user interaction. The core of the methodology is a dual-LLM engine [18] that leverages the specific strengths of different models to achieve high logical accuracy and perfect syntactical compliance. This separation of concerns at both the architectural and model levels is intended to improve robustness, scalability, and reproducibility of the generation process.

A. System Architecture and Workflow

The end-to-end workflow of AIDTG is designed to integrate seamlessly into an agile QA process, moving from high-level requirements to executable scripts. The overall system architecture follows a component-based design and is represented using a C4 Container diagram, as illustrated in Fig. 1.

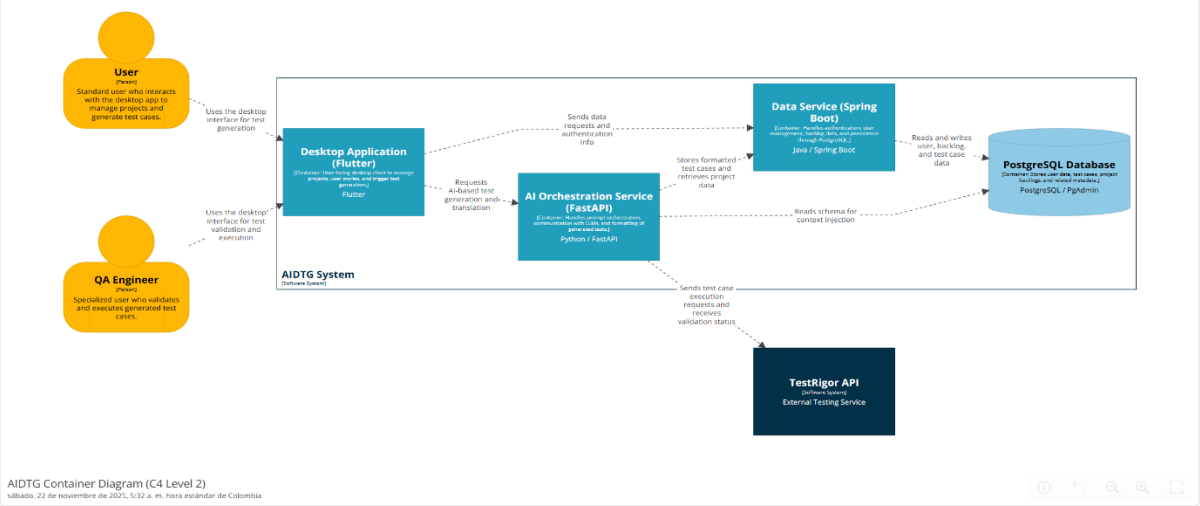

Fig. 1. AIDTG C4 Container S8.

The workflow proceeds through a series of well-defined stages, each corresponding to a distinct responsibility within the test generation lifecycle:

1) Project and backlog management: The user first creates a "Project" within the desktop application. Into this project, the user injects the Product Backlog, consisting of "Epics" and their corresponding "User Stories" (US) in a structured format (e.g., CSV). This structure enables traceability between backlog items and generated test cases, which is essential for agile quality assurance practices.

2) Test cycle initiation: The user selects specific User Stories from the Product Backlog to form a "Release Backlog" for the current test cycle. The user then initiates the generation script for this selected backlog. This step mirrors real-world sprint or release planning activities, ensuring that the generated test cases align with the scope of the current development iteration.

3) Context injection: At the moment of generation, the system prompts the user to provide the "frozen database schema" (e.g., a .sql script). This is the critical contextual step that makes the framework data-aware, informing the generation process with the project's actual data model [2]. The term "frozen" denotes that the schema represents a stable snapshot of the data model for the duration of the test cycle, ensuring consistency across generated artifacts.

4) Orchestration and generation: The backend orchestrator (detailed in 4.4) receives the User Stories and the DB schema. It combines them into a master prompt and executes the Dual-LLM workflow [detailed in Section IV(C)]. At this stage, prompt engineering rules, coverage strategies, and contextual constraints are jointly applied to guide the generation process.

5) Output and translation: The system generates the test cases and provides two distinct outputs: a) a translation into the user's native language (e.g., Spanish) for human validation, and b) a script formatted in TestRigor BDD 2.0 syntax [23].

This dual-output strategy supports both Human-in-the-Loop review and immediate machine execution.

6) Optional execution and monitoring: The user has the option to execute the generated BDD 2.0 script directly via an API call to the TestRigor platform. If this option is chosen, the system utilizes WebSockets to maintain a persistent connection, monitoring the execution status in real-time and reporting "Pass" or "Fail" results. This optional execution step enables immediate validation of syntactic correctness and executability on the target testing platform.

7) Human-in-the-loop feedback: Finally, the user is prompted to rate the quality and correctness of the generated test case on a 1-to-5-star scale. This feedback mechanism is crucial for the evaluation phase of our study (Section V). Although not used for automated learning in this study, this feedback channel establishes the foundation for future Human-in-the-Loop refinement mechanisms.

8) Dataset description: To ensure transparency and reproducibility, the experimental dataset used in this study is described explicitly.

The dataset consists of a real-world Product Backlog obtained from an active software development project within an industrial context. It includes 50 distinct User Stories distributed across 8 Epics, representing common functional requirements in a business-oriented information system.

Each User Story contains a textual description, priority level, acceptance criteria, and estimated story points, providing sufficient detail for acceptance-level test design.

This dataset was selected to reflect realistic agile development conditions, including heterogeneous requirement complexity and domain-specific constraints.

B. Core Technologies

The system is implemented as a decoupled application. The backend is a high-performance Python API, while the frontend is a modern desktop application.

The backend is built using FastAPI, a high-performance web framework chosen for its asynchronous capabilities, essential for managing concurrent API calls to LLM endpoints. It runs on a Uvicorn ASGI server. For data persistence, SQLAlchemy is used as the ORM to manage all projects, backlogs, and generated test cases, connecting to a PostgreSQL database via the psycopg2 driver. User authentication and security are handled using python-jose for JWT token generation and Passlib with bcrypt for secure password hashing. These technologies were selected to ensure scalability, security, and efficient handling of concurrent generation and execution requests.

The core backend technologies are detailed in Table I.

TABLE I. BACKEND LIBRARIES

Library / Technology	Description
FastAPI	Modern web framework for building asynchronous APIs.
Uvicorn	ASGI server for running the FastAPI application.
SQLAlchemy	Python SQL Toolkit and ORM for database interaction.
psycopg2-binary	PostgreSQL adapter for Python.
Pandas / Openpyxl	Used for data manipulation and parsing of input data.
OpenAI	Python client library for accessing the GPT-4.0 API.
google-generativeai	Python client library for accessing the Gemini 1.5 Pro API.
Tiktoken	Used for token counting to manage OpenAI prompt length and costs.
python-dotenv	Manages environment variables (e.g., API keys).
python-jose & Passlib	Libraries for implementing OAuth2 security, token generation, and password hashing.
fastapi-pagination	Handles pagination for API endpoints returning large lists (e.g., backlog items).
httpx	Asynchronous HTTP client for making robust API calls to LLM endpoints.

The frontend is a desktop application built with React and Next.js, providing a responsive and modern user interface. Tailwindcss and Next-UI are used for styling and components, while React-Markdown is used to render the table-formatted outputs from the LLM. This technology stack enables clear visualization of generated test cases and seamless interaction with the orchestration backend.

The core desktop technologies are detailed in Table II.

TABLE II. DESKTOP LIBRARIES

Library / Technology	Description
Next / React	Core JavaScript libraries for building the user interface.
React-dom	Entry point to the DOM and server renderers for React.
React-markdown	React component to render the Markdown output from the LLM.
Next-ui	A component library for building beautiful UIs with Next.js.
Tailwindcss	A utility-first CSS framework for rapid UI development.
Github-markdown-css	CSS for replicating GitHub's Markdown-rendered style.

C. The Dual-LLM Orchestration Workflow

A core innovation of AIDTG is its "dual-engine" orchestration pipeline, which uses two different state-of-the-art LLMs in sequence. This design is conceptually aligned with emerging dual-model strategies reported in recent AI research, where task specialization across models improves overall output quality.

Step 1: Generation (LLM-1: Gemini 1.5 Pro)

The first engine is the "Generation Engine." It uses Gemini 1.5 Pro for its large context window and strong reasoning capabilities. It receives the most complex input: the "master prompt" (detailed in 4.4), which contains the User Stories and the full database schema. Its responsibility is limited to logical reasoning and test design, deliberately excluding strict syntax enforcement.

Step 2: Translation and Formatting (LLM-2: GPT-4.0)

The structured Markdown output from Gemini is then passed to GPT-4.0, which acts as a specialized "Formatting and Translation Engine". This model executes two distinct tasks via separate, simpler prompts:

1) Native language translation: It translates the Markdown table into the user's native language (e.g., Spanish) for human validation.

2) BDD 2.0 formatting: It translates the same Markdown table into the strict, proprietary BDD syntax required by TestRigor BDD 2.0.

This separation of logical reasoning from syntactic formatting significantly reduces the risk of syntax-related hallucinations.

D. Core Component: The AIDTG Prompt Engineering Framework

The primary intellectual property of the AIDTG framework is its Master Prompt Template, which is dynamically constructed by the backend orchestrator and sent to the Generation Engine (Gemini 1.5 Pro). Rather than relying on a single monolithic instruction, the prompt is composed of multiple semantically distinct components, each controlling a specific aspect of LLM behavior.

This subsection deconstructs the master prompt to explain the design rationale behind each component.

1) Persona and task injection: The prompt's first section, shown in Fig. 2, is the Persona Injection. This is the most critical step for setting the context, quality bar, and expertise of the model [12]. Instead of a generic instruction, the prompt forces the LLM to adopt the role of a Senior QA Engineer and explicitly names the advanced methodologies it must use, such as Risk-Based Testing [2] and Combinatorial Testing. This immediately constrains the model to a professional context and significantly improves the quality and relevance of the generated test cases.

```
You are a **Senior QA Engineer** specialized in **Risk-Based Testing** and **Combinatorial Testing Techniques (Pairwise Testing)**.
---task---
Your task is to **generate a table of Test Cases (TC)** in **Markdown format**, based on the following **User Stories (US)** and the provided **contextual database**. Do not include explanations or commentary – only return **a single Markdown table** with the required fields.
```

Fig. 2. Persona injection prompt.

2) Multi-language and domain handling: The second component, detailed in Fig. 3, addresses the challenge of internationalization. The prompt explicitly instructs the LLM to auto-detect the language of the input User Story and generate all test cases in that same language. This makes the framework immediately usable for global teams (e.g., in English, Spanish, Hindi) without modification. It also mandates the preservation of domain-specific terms (like currencies or local names), preventing the LLM from "over-translating" and losing critical context.

```
### Language Handling
For each User Story:
**Automatically detect its language** (e.g., English, Spanish, Chinese, Hindi, etc.).
**Generate all Test Cases in the same detected language.**
Preserve any culturally specific or domain-related terms (such as currencies, location names, etc.) without translation.
```

Fig. 3. Multi-language injection prompt.

3) Contextual data injection (Schema-Awareness): The third component, shown in Fig. 4, is the core of the orchestration [11]. The backend dynamically injects two critical data blocks: the User Stories (functional context) and the Database Schema (technical context).

a) User story data: The {user_epics_prompt} variable passes the selected backlog items in a machine-readable CSV format, which the LLM is told how to parse.

b) Frozen database: The {db_prompt} variable provides the "ground truth" for the test data. By providing real input values (e.g., product types, currencies, locations), we move the LLM from generating plausible tests to generating realistic and executable tests grounded in the system's actual data model.

```
### User Story Data
User Stories are represented in **CSV format**, delimited by the character `|`.
### Columns:
ID | DESCRIPTION | PRIORITY | STORY_POINTS | ACCEPTANCE_CRITERIA | PAIR_WISE_TESTING`
--- START OF USER STORY DATA ---
{user_epics_prompt}
--- END OF USER STORY DATA ---
### Frozen Database (Testing Context)
The following dataset can be used as **real input values** to design your Test Cases
(for example: product types, currencies, or locations).
--- START OF FROZEN DATABASE DATA ---
{db_prompt}
--- END OF FROZEN DATABASE DATA ---
```

Fig. 4. Context data injection prompt.

4) Strict output formatting (Machine-Readability): A primary challenge in LLM pipelines is output unreliability [17]. To solve this, the fourth component (Fig. 5) enforces a strict, non-negotiable output format. The prompt commands the LLM to return only a single Markdown table with a precisely defined set of headers. This "structured-to-structured" pipeline (CSV/DB -> Markdown) is the key to reliability. It ensures the output from the Generation Engine (Gemini) is a machine-readable artifact, which can be perfectly and safely parsed by the Formatting Engine (GPT-4.0) in the next step, eliminating the risk of free-text "hallucinations" or conversational contamination.

```
### REQUIRED OUTPUT STRUCTURE
Return a single **Markdown table** with the following headers:
### Columns:
| ID | (HU_FAKE_ID) | (HU_ID) | OBJECTIVE | TYPE | PRIORITY | PRE-CONDITION | INPUTS | STEPS | EXPECTED RESULTS | POST-CONDITION |
```

Fig. 5. Output formatting prompt.

5) Test case generation and strategy rules: Finally, the fifth component (Fig. 6) provides the explicit "business logic" for the generation.

Rules 1 & 2 (Coverage & Traceability): Mandate the minimum required coverage (Happy, Error, and Alternative Paths) and ensure traceability by mapping the test case back to the User Story ID.

Rules 3 & 4 (Quality): Instruct the LLM to use the provided database data for Inputs and to write specific, verifiable Expected Results.

Rule 5 (Advanced Strategy): This is the most complex rule. It activates the Pairwise Testing persona if the PAIR_WISE_TESTING flag is True. It commands the LLM to look at the database values (e.g., Product Type, Currency, Location) and create an efficient set of test cases that maximize combinatorial coverage with minimal tests. This transforms the LLM from a simple generator into an intelligent test strategist.

```
### TEST CASE GENERATION RULES
1. **Minimum Coverage:**
   For each US, generate TCs covering:
   - **Happy Path:** main successful flow.
   - **Error Path:** validations and incorrect inputs.
   - **Alternative Path:** optional or exceptional flows
2. **Traceability:**
   Use the **ID** field from the US as the value for **(HU_ID)** in the table.
3. **Inputs and Steps:**
   Be detailed and realistic. Use actual data from the provided database whenever possible.
4. **Expected Results:**
   Must be specific and verifiable (e.g., "An error message is displayed: 'Invalid email format'").
5. **Priority:**
   Inherit directly from the User Story (Critical, High, Medium, Low).
6. **Pairwise Testing (if applicable):**
   If the flag `PAIR_WISE_TESTING = True` is active for a US:
   - Generate **test combinations** using real values found in the database section.
   - Example for a "Add Product" US:
     - Product Type (physical, digital)
     - Currency (PEN, USD)
     - Location (Lima, Cusco, Arequipa)
   - Create **unique, efficient test cases** that maximize coverage with minimal combinations.
7. **Formatting Rules:**
   - Only one Markdown table.
   - No text outside the table.
   - Use "Happy Path", "Error Path", or "Alternative Path" under "TYPE (Path)".
   - Fill all columns completely (no empty cells).
   - Separate each step in the **STEPS** column using a line break `\\n`.
   - Maintain the same language as the detected User Story.
```

Fig. 6. Test case generation rules prompt.

V. Results

This section details the experimental design used to validate this research's objectives. First, a framework of evaluation metrics is defined based on academic literature. Second, the experimental setup is described. Finally, the quantitative and qualitative results obtained from the application of the AIDTG framework on a real-world dataset are presented and discussed.

The results are structured around the predefined research questions to ensure traceability between objectives, evaluation metrics, and empirical findings.

A. Definition of Evaluation Metrics

To evaluate the efficacy and performance of a new test generation technology, a metrics framework is essential. Based on a review of the literature on test automation [31, 2132, 35], technology adoption [35], and Natural Language Processing (NLP) in testing [33, 34], we have identified a consensus around four key performance variables. These variables are widely used in both academic studies and industrial evaluations of automated testing tools.

Table III summarizes these standard industry and academic metrics.

TABLE III. Key Literature-based Metrics

Metric / Variable	Description	Rationale (from Literature)
Efficiency / Time Savings	Measures the reduction in human time and effort (measured in man-minutes) required to complete a task, compared to the manual baseline.	The primary justification for adopting new testing technologies is operational efficiency, cost reduction, and accelerating delivery cycles [35, 34].
Perceived Quality & Precision	Measures how correct, readable, relevant, and useful the generated artifacts are. This is a qualitative metric best measured via human expert scoring (surveys).	A systematic review of NLP in testing [33] identifies "Test Case Quality" as a central metric for validation.

Functional Correctness	Measures whether the generated test cases are syntactically correct and capable of executing on a testing platform without failure.	This is the ultimate technical validation. Rapid generation is useless if the scripts are unexecutable on the target platform [35].
Fault Detection	Measures the ability of the generated test cases to identify defects (bugs) in the codebase.	The ultimate goal of testing is to find faults [32]. High fault detection indicates high-quality generation.

For this study, we focused on the first three metrics (Efficiency, Perceived Quality, and Functional Correctness), as "Fault Detection" is dependent on pre-existing bugs in the source code, which was outside the scope of our backlog-based generation.

This scope delimitation ensures that the evaluation remains aligned with the proactive nature of acceptance test generation from requirements.

B. Experimental Setup

The experiment was designed to compare the performance of the AIDTG tool against traditional manual generation.

1) Dataset: We used a real-world project dataset from a software development company (identified as the organization under study). The dataset consisted of a Product Backlog with 50 unique User Stories (US), distributed across 8 Epics. This dataset corresponds exactly to the one described in the Methodology section, ensuring internal consistency across the study.

2) Control group (Manual): We measured the average time it took for a QA analyst with 3 years of experience (hereafter, "the expert") to analyze a US, design, and manually write the test cases (Happy, Alt, Error) based on the requirements. This process represents standard industry practice for acceptance test design in agile teams.

3) Experimental group (AIDTG): We used the AIDTG framework (configured with Gemini 1.5 Pro and GPT-4.0) to process the same 50 US.
A group of five QA analysts (with mixed experience levels) reviewed and rated the generated test cases. Their evaluations were averaged to ensure inter-rater reliability. The use of multiple evaluators strengthens the robustness of the qualitative assessment.

4) Instruments:

a) Efficiency: Time tracking (man-minutes) for both groups.

b) Perceived quality: An evaluation survey (see Appendix A) based on a 1-to-5 Likert scale (1=Useless, 5=Fully Adequate), which the expert completed for each generated test case. The survey instrument was designed to capture expert judgment on relevance, clarity, and practical usability.

c) Functional correctness: The TestRigor platform, used to execute the BDD 2.0 scripts and report a binary ("Pass" / "Fail") result.

C. Research Questions

The experiment sought to answer the following Research Questions (RQs), derived from the project's specific objectives:

1) RQ1 (Efficiency): To what extent does AIDTG reduce the time and effort of test case creation compared to the manual process?

2) RQ2 (Quality): How precise, useful, and of high quality are the test cases generated by AIDTG, according to human expert evaluation (based on the survey)?

3) RQ3 (Functional correctness): What percentage of the BDD 2.0 scripts generated by AIDTG are functionally correct and executable ("Pass") on the TestRigor platform?

These research questions directly map to the selected evaluation metrics, enabling a clear interpretation of results.

D. Results and Findings

The 50 User Stories were processed through the AIDTG pipeline, resulting in the generation of 197 unique test cases (covering Happy, Alternative, and Error paths). This distribution reflects the minimum coverage rules enforced by the generation framework.

1) Results for RQ1 (Efficiency): A significant reduction in effort was observed. The manual process required an average of 25 minutes per User Story. The AIDTG process (Table IV), including AI generation and human review, reduced the total effort to approximately 5 minutes per User Story.

TABLE IV. COMPARISON BETWEEN MANUAL PROCESS AND AIDTG PROCESS

Metric	Manual Process (Control)	AIDTG Process (Experimental)	Improvement
Avg. Time per US	25 Minutes	5 Minutes	80% Time Savings
Total Effort (50 US)	1250 Minutes (20.8 hrs)	250 Minutes (4.1 hrs)	16.7 hours saved

Conclusion (RQ1): The AIDTG framework achieved an 80% reduction in the time and effort required for test case creation.

2) Results for RQ2 (Perceived quality): The evaluation survey (Appendix A) was used to measure the perceived quality of the 197 generated TCs. Table V shows the distribution of the expert's ratings, with qualitative labels removed as requested.

TABLE V. SCALE RANKING

Rating (1-5 Scale)	# of Generated TCs	Percentage
5	158	80.2%
4	29	14.7%
3	10	5.1%
2	0	0%
1	0	0%

Conclusion (RQ2): 94.9% of the generated test cases received a rating of 4 or 5, resulting in a weighted average score of 4.75 out of 5.

3) Results for RQ3 (Functional correctness): To validate technical correctness, all 197 BDD 2.0 scripts generated by the GPT-4.0 engine were sent to the TestRigor platform (via WebSocket) for syntax validation. The results are summarized in Table VI.

TABLE VI. COLLECTED METRICS

Metric	# of TCs	Percentage	Analysis Notes
"Pass" (Successful Execution)	181	91.9%	BDD 2.0 syntax is correct and executable.
"Fail" (Syntax Error)	16	8.1%	Execution failed due to syntax.
Total	197	100%	

Conclusion (RQ3): 91.9% of the generated scripts were syntactically correct and executable on the target platform.

E. Discussion of Results

The results strongly validate the project's hypothesis. An 80% time saving (RQ1) aligns with industry promises of technology adoption [35] and exceeds the expectations of traditional NLP tools that require more intensive setup [34].

The most significant finding is the synergy between high efficiency (RQ1) and high quality (RQ2). Historically, test generation tools sacrificed quality for speed [32]. The AIDTG framework, by using a dual-engine (Gemini for logic, GPT-4 for format) and a DB-schema-aware orchestration, demonstrates that it is possible to achieve both. The 4.75 average rating (RQ2) and the 91.9% execution success rate (RQ3) prove that the output is not just "fast," but "correct and useful."

While a 1–5 star rating function was implemented in the application [as detailed in Section IV(A)], it is considered an experimental feature for future Human-in-the-Loop feedback. It is important to emphasize that the data for this study was collected exclusively via formal surveys (detailed in Appendix A: QA Evaluation Questionnaire) to ensure methodological rigor.

VI. CONCLUSION AND FUTURE WORK

Specifically, the study focused on the challenges of translating high-level Product Backlog requirements into executable acceptance tests in a manner that is both efficient and semantically accurate. [6]. While recent LLM research has focused on unit tests [5] and bug reproduction [7], a significant gap remained in the proactive, high-level generation of acceptance tests from backlog requirements.

This gap is particularly critical in agile and DevOps contexts, where acceptance testing plays a central role in validating business requirements prior to release.

We successfully designed, implemented, and evaluated the AI-Based Desktop Test Generator (AIDTG), an LLM orchestration framework that bridges this gap. The proposed framework moves beyond isolated prompt-based experimentation by introducing an end-to-end, schema-aware orchestration pipeline for acceptance test generation.

Our methodology's primary contribution is a dual-LLM engine (Gemini 1.5 Pro and GPT-4.0) combined with a schema-aware prompt engineering framework (Section IV). This combination enables a clear separation between logical test design and strict syntactic formatting, improving both generation quality and executability.

This approach transforms high-level User Stories into executable TestRigor BDD 2.0 scripts [23] by grounding the generation process in the project's actual data model [2]. By explicitly incorporating database constraints into the generation workflow, the framework reduces ambiguity and increases semantic alignment between requirements and validation artifacts.

Our experimental evaluation on a real-world dataset demonstrated that AIDTG:

- Reduces test design effort by 80% (RQ1).
- Achieves a 4.75 out of 5 average quality rating from expert human review (RQ2).
- Produces BDD scripts with a 91.9% functional (syntactical) correctness rate on the target platform (RQ3).

These results confirm that the proposed framework can significantly improve QA productivity while maintaining high standards of test quality and executability.

Unlike prior studies that evaluate prompt effectiveness in isolation, this research provides empirical evidence for the benefits of orchestration-centric architectures in AI-assisted software testing.

From an industrial perspective, the findings suggest that AIDTG can support the adoption of AI-assisted testing practices in real agile teams.

By automating repetitive acceptance test design tasks, QA professionals can focus on higher-value activities such as exploratory testing, test strategy refinement, and defect analysis.

Furthermore, the generation of business-readable BDD artifacts has the potential to improve communication and alignment between developers, testers, and non-technical stakeholders.

A. Future Work

This study opens several avenues for future research.

- Mitigating Failures: The 8.1% failure rate in functional correctness (RQ3) was primarily due to minor syntax hallucinations by the formatting LLM. Future work will focus on refining the formatting prompt and implementing a "self-correction" loop where the LLM automatically fixes the script upon receiving a "Fail" status from the TestRigor API.

- Human-in-the-Loop (HIL) Fine-Tuning: The 1-5 star rating feature built into the application (Section IV) was experimental for this study. The next phase is to capture this user feedback and use it to create a fine-tuning dataset, enabling a Human-in-the-Loop (HIL) pipeline that constantly improves the generation engine's accuracy.
- Expanding Target Frameworks: While this work focused on TestRigor BDD 2.0 [23], the dual-engine methodology is adaptable. Future iterations could include formatting engines for other popular BDD frameworks, such as Cucumber (Gherkin) or Behave.
- Integration with Design Tools: A promising avenue is to expand the "context" beyond just the database schema to include inputs from UI/UX design tools (e.g., Figma), further grounding the generated test cases in the application's intended design.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of the Peruvian University of Applied Sciences (UPC) and its Directorate of Research.

REFERENCES

[1] Brynjolfsson, E., Li, D., & Raymond, L. (2025). Generative AI at Work. The Quarterly Journal of Economics, 140(2), 889–942. https://doi.org/10.1093/qje/qjae044.

[2] Felderer, M., & Schieferdecker, I. (2014). A taxonomy of risk-based testing. International Journal on Software Tools for Technology Transfer. https://doi.org/10.1007/s10009-014-0332-3.

[3] Islam, M., Khan, F., Alam, S., & Hasan, M. (2023, September). Artificial Intelligence in Software Testing: A Systematic Review. In TENCON 2023 - 2023 IEEE Region 10 Conference (TENCON). IEEE. https://doi.org/10.1109/TENCONS8879.2023.10122349.

[4] Kang, S., Yoon, J., & Yoo, S. (2023). Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (pp. 2312–2323). IEEE. https://doi.org/10.1109/ICSE48619.2023.00194.

[5] Ouédraogo, W. C., Kaboré, K., Song, Y., Klein, J., Tian, H., Koyuncu, A., Lo, D., & Bissyandé, T. F. (2024). LLMs and Prompting for Unit Test Generation: A Large-Scale Evaluation. In 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24). ACM. https://doi.org/10.1145/3691620.3695330.

[6] Panwar, A., & Peddi, P. (2023). Challenges in Software Testing. International Journal of Renewable Energy Exchange, 11(1), 168–171.

[7] Plein, L., Ouédraogo, W. C., Klein, J., & Bissyandé, T. F. (2024). Automatic Generation of Test Cases based on Bug Reports: a Feasibility Study with Large Language Models. In 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24). ACM. https://doi.org/10.1145/3639478.3643119.

[8] Santos, R., Santos, I., Magalhaes, C., & de Souza Santos, R. (2024). Are We Testing or Being Tested? Exploring the Practical Applications of Large Language Models in Software Testing. In 2024 IEEE Conference on Software Testing, Verification and Validation (ICST) (pp. 353–360). IEEE. https://doi.org/10.1109/ICST60714.2024.00039.

[9] Stryker, C. (2024, May 15). What is AI? IBM Think. https://www.ibm.com/think/topics/artificial-intelligence.

[10] Stryker, C. (2024, March 19). What are LLMs? IBM Think. https://www.ibm.com/think/topics/large-language-models.

[11] Winland, V., & Noble, J. (2024, June 25). What is LLM Orchestration? IBM Think. https://www.ibm.com/think/topics/llm-orchestration.

[12] (2024, May 15). What is prompt engineering? IBM Think. https://www.ibm.com/think/topics/prompt-engineering.

[13] Karpurapu, S., Myneni, S., et al. (2024). Comprehensive Evaluation... of Large Language Models in the Automation of Behavior-Driven Development... IEEE Access. https://doi.org/10.1109/ACCESS.2024.3391815.

[14] Lim, J. W., Chiew, T. K., et al. (2024). Test case information extraction from requirements specifications using NLP-based unified boilerplate approach. The Journal of Systems and Software. https://doi.org/10.1016/j.jss.2024.112005.

[15] Dakhel, A. M., Nikanjam, A., et al. (2024). Effective test generation using pre-trained Large Language Models and mutation testing. Information and Software Technology. https://doi.org/10.1016/j.infsof.2024.107468.

[16] Yang, R., Xu, X., & Wang, R. (2025). LLM-enhanced evolutionary test generation for untyped languages. Automated Software Engineering. https://doi.org/10.1007/s10515-025-00496-7.

[17] Li, Y., Liu, P., et al. (2025). Evaluating large language models for software testing. Computer Standards & Interfaces. https://doi.org/10.1016/j.csi.2024.103942.

[18] Zhuge, Q., Wang, H., & Chen, X. (2025). TwinStar: A Novel Design for Enhanced Test Question Generation Using Dual-LLM Engine. Applied Sciences. https://doi.org/10.3390/app1506305.

[19] Alagarsamy, S., Tantithamthavorn, C., & Aleti, A. (2024). A3Test: Assertion-Augmented Automated Test case generation. Information and Software Technology. https://doi.org/10.1016/j.infsof.2024.107565.

[20] Pasca, E. M., Delinschi, D., et al. (2025). LLM-Driven, Self-Improving Framework for Security Test Automation... Leveraging Karate DSL... IEEE Access. https://doi.org/10.1109/ACCESS.2025.3554960.

[21] Mehmood, A., Ilyas, Q. M., et al. (2024). Test Suite Optimization Using Machine Learning Techniques: A Comprehensive Study. IEEE Access. https://doi.org/10.1109/ACCESS.2024.3490453.

[22] Chen, J., Chen, Y., et al. (2024). DynER: Optimized Test Case Generation for Representational State Transfer (REST)ful... Electronics. https://doi.org/10.3390/electronics13173476.

[23] TestRigor. (n.d.). TestRigor - #1 Generative AI-based Test Automation Tool. Consultado el 8 de noviembre de 2025. https://testrigor.com/.

[24] TestRigor. (2023, 11 de diciembre). Revolutionizing QA: How to Create Tests in Seconds with testRigor's Generative AI. testRigor Blog. https://testrigor.com/blog/revolutionizing-qa-how-to-create-tests-in-seconds-with-testrigors-generative-ai/.

[25] Fernández Del Carpio, A., Bermon Angarita, L., & Osorio Londoño, A. A. (2022). A Bibliometric Analysis of DevOps Metrics. DESIDOC Journal of Library & Information Technology. https://doi.org/10.14429/djlit.42.6.18365.

[26] Rehan, S., Al-Bander, B., & Al-Said Ahmad, A. (2025). Harnessing Large Language Models for Automated Software Testing: A Leap Towards Scalable Test Case Generation. Electronics. https://doi.org/10.3390/electronics14071463.

[27] Ouédraogo, W. C., Plein, L., Kaboré, K., Habib, A., Klein, J., Lo, D., & Bissyandé, T. F. (2025). Enriching automatic test case generation by extracting relevant test inputs from bug reports. Empirical Software Engineering. https://doi.org/10.1007/s10664-025-10635-z.

[28] Christakis, N., & Drikakis, D. (2025). Evaluating Large Language Models in Code Generation: INFINITE Methodology for Defining the Inference Index. Applied Sciences. https://doi.org/10.3390/app15073784.

[29] Ferrari, F. C., Durelli, V. H. S., Andler, S. F., Offutt, J., Saadatmand, M., & Müllner, N. (2023). On Transforming Model-based Tests into Code: A Systematic Literature Review. Software Testing, Verification and Reliability. https://doi.org/10.1002/stvr.

[30] Khaliq, Z., Farooq, S. U., & Khan, D. A. (2022). A deep learning-based automated framework for functional User Interface testing. Information and Software Technology. https://doi.org/10.1016/j.infsof.2022.106969.

[31] Karimi, M., Kolahdouz-Rahimi, S., & Troya, J. (2024). Ant-colony optimization for automating test model generation... The Journal of Systems and Software, 208. https://doi.org/10.1016/j.jss.2023.111882.

[32] Qin, Z., Fan, J., Liu, X., Li, Z., & Sun, X. (2025). Effective fuzzing testcase generation based on variational auto-encoder... Engineering

Applications of Artificial Intelligence, 144. https://doi.org/10.1016/j.engappai.2025.110094.

[33] Boukhlif, M., Hanine, M., Kharmoum, N., Ruigómez Noriega, A., García Obeso, D., & Ashraf, I. (2024). Natural Language Processing-Based Software Testing: A Systematic Literature Review. IEEE Access, 12. https://doi.org/10.1109/ACCESS.2024.3407753.

[34] Koh, S. J., & Chua, F. F. (2023). ReqGo: A Semi-Automated Requirements Management Tool. International Journal of Technology, 14(4). https://doi.org/10.14716/ijtech.v14i4.6151.

[35] Poth, A., Rrjolli, O., & Arcuri, A. (2025). Technology adoption performance evaluation applied to testing industrial REST APIs. Automated Software Engineering, 32(5). https://doi.org/10.1007/s10515-024-00477-2.

APPENDIX A. QA EVALUATION QUESTIONNAIRE

This appendix presents the Quality Assurance (QA) evaluation questionnaire used in the study. The purpose of this questionnaire is to gather expert feedback on the usefulness, reliability, and clarity of the test cases generated by the AI-based solution, as well as the feasibility of integrating this tool into agile software development processes. The questionnaire is divided into three parts: Section A contains Likert-scale items for quantitative evaluation, Section B provides dichotomous (Yes/No) questions for binary assessment, and Section C includes open-ended questions for qualitative feedback.

Instructions to Respondents:

- Please answer all questions based on your experience with the AI-generated test cases and the overall tool.
- In Section A, rate each statement on a scale of 1 to 5 (where 1 represents Very Poor and 5 represents Excellent).
- In Section B, select either Yes or No for each question.
- In Section C, provide your answers in your own words, elaborating on your perspective for each question.

Section A – Likert-Scale Questions (1–5):

-How useful do you consider the AI-generated test cases compared to manual test cases?

-What level of reliability do you perceive in the AI-generated results with respect to the user requirements?

-On a scale of 1 to 10, how accurate do you consider the test cases generated by AI?

-On a scale of 1 to 10, how complete do you consider the set of critical scenarios identified by AI?

-How precise do you consider the critical scenarios generated by AI?

-How likely are you to recommend this solution for a real software development project?

-**Scale:** 1 = Very Poor, 5 = Excellent.

Section B – Dichotomous Questions (Yes/No):

-Do you believe that automatic test case generation with AI can significantly reduce test design time?

-Do you think this solution can be seamlessly integrated into agile methodologies such as Scrum or Kanban?

-Do you consider the AI-generated test cases to be sufficiently clear for execution by a human tester?

Section C – Open-Ended Questions:

-What aspects do you consider most valuable about the automatic generation of test cases with AI?

-What are the main limitations or risks you identify in the application of this solution?

-What improvements would you recommend to increase the effectiveness of the tool in real-world testing scenarios?

-How do you envision the impact of this solution on the future of software quality assurance?