American University Kyiv

A Capstone Project

AI INTEGRATION FOR TEST CASES GENERATION AND MAINTENANCE: OPTIMIZING TEST TEAM WORKFLOW

ІНТЕГРАЦІЯ ШІ ДЛЯ ГЕНЕРАЦІЇ ТА СУПРОВОДУ ТЕСТ-КЕЙСІВ: ОПТИМІЗАЦІЯ РОБОЧИХ ПРОЦЕСІВ КОМАНДИ ТЕСТУВАННЯ

by Student’s Taras Stepaniuk

Presented in Partial Fulfillment of the Requirements
for the Degree

Master

APPROVED BY:

Professor Viktor Putrenko, Ph.D, Faculty Mentor

2026

ABSTRACT

Manual quality assurance workflows in fast-paced software delivery increasingly struggle to keep pace with rapid code evolution, with QA teams spending disproportionate effort on interpreting requirements and maintaining test documentation. This research investigates whether an Artificial Intelligence-driven middleware can optimize this workflow by automating the synthesis of requirements, design, source code, and existing test documentation into actionable testing artifacts.

A middleware solution was designed and implemented on the Fastlane framework, integrating data from Asana, Figma, GitLab, and TestRail, and leveraging the OpenAI GPT-4.1 model through a structured Chain-of-Thought prompt. The evaluation combined quantitative KPI tracking across 24 production tasks with qualitative feedback from three QA engineers.

The results demonstrate a 68% reduction in the test documentation effort ratio, a decrease in the median number of dev/test iterations from three to one, and a doubling of the single-iteration resolution rate. Qualitative analysis confirmed accelerated feature comprehension and reduced cognitive load. The study validates a hybrid human–AI model of quality assurance and defines a roadmap toward autonomous test maintenance.

Keywords: artificial intelligence, software testing, test case generation, quality assurance, CI/CD, large language models, middleware

TABLE OF CONTENTS

CHAPTER 1. THEORY OVERVIEW

The modern software engineering landscape is characterized by increasing complexity and the demand for rapid delivery cycles. To meet these challenges, the industry is shifting away from purely manual processes toward intelligent automation. The theoretical foundation of this research lies in the intersection of traditional software development methodologies and advanced Artificial Intelligence (AI) techniques.

This chapter provides a comprehensive overview of how AI technologies are currently integrated into the Software Development Life Cycle (SDLC). It examines the evolution of quality assurance practices, moving from static manual testing to dynamic, AI-driven strategies. By analyzing current methodologies in requirement processing, code analysis, and automated test maintenance, this section establishes the theoretical necessity for a middleware solution that synchronizes project management tools with technical environments. Understanding these theoretical pillars is essential for developing a robust framework for automated test case generation and long-term documentation sustainability.

1.1 Aim, goals, object and subject of the research

This research addresses the necessity of bridging the gap between project management tools and testing repositories by leveraging Artificial Intelligence (AI). By introducing an intelligent middleware that integrates data from Asana and GitLab, the study proposes a shift-left approach to automated test documentation generation.

The aim of the research is to design and implement an AI-driven middleware solution that automates the generation of testing documentation (checklists and test cases) by analyzing requirements and application source code, thereby reducing manual overhead and improving the technical accuracy of the testing process.

To achieve this aim, the following research tasks have been identified:

- To analyze the current state of AI integration in the Software Development Life Cycle (SDLC) and its role in quality assurance.
- To investigate the challenges of manual testing workflows within a startup project ecosystem involving Asana, GitLab, and TestRail.

- To develop a theoretical and architectural framework for a middleware that utilizes Natural Language Processing (NLP) and code analysis for test generation.
- To implement the proposed solution and integrate it into the existing team workflow.
- To evaluate the impact of the middleware using quantitative KPIs and qualitative team feedback.

The object of the research is the software quality assurance process within the agile development lifecycle of a startup product.

The subject of the research is the methods and tools for automated testing documentation generation based on the synthesis of project management requirements and application source code through AI-driven middleware.

1.2 Software Development Life Cycle overview

SDLC — is the activities performed at each phase in software development, and how they relate to one another logically and chronologically. [3]

Defining the specific SDLC stages ensures that development is organized and executed effectively, resulting in high-quality software that meets user requirements. [16] SDLC usually consists of 7 phases.

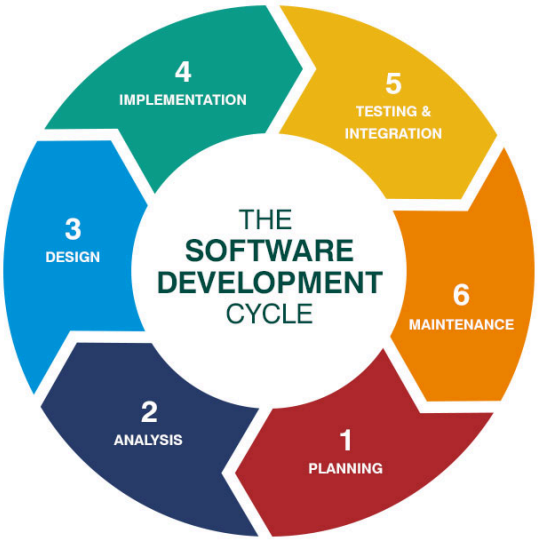

Figure 1. Graph showing the phases of SDLC. Source: [17]

1. Planning: project goals, objectives and requirements are defined and documented.
2. Feasibility analysis: if the project is technically and financially viable.
3. System design: creating the system design and architecture.
4. Implementation: it is a development phase, when the plans are transformed into a functional system.
5. Testing: the goal of this phase is to receive the feedback about the system created, identify and fix issues.
6. Deployment: after the system is ready it is being delivered to the end-users.
7. Maintenance: it is an ongoing support of the system, so that is remains working and relevant over time.

1.3 AI in Software Development Life Cycle

The integration of Artificial Intelligence into the Software Development Life Cycle (SDLC) is fundamentally reshaping how software products are conceived, built, and maintained. Rather than following traditional, rigid linear paths, AI introduces intelligent automation and predictive capabilities across every phase of development, fostering a more adaptive and data-driven environment [2].

Key Impact Areas across SDLC Phases:

- Requirement Analysis & Planning: AI leverages Natural Language Processing (NLP) to automate the extraction of requirements and identify inconsistencies or ambiguities in documentation. This significantly reduces communication gaps between stakeholders and technical teams early in the cycle [2].
- Design & Architecture: AI algorithms assist in generating architectural models and UI prototypes. By analyzing historical project data and usability standards, AI optimizes design patterns to ensure scalability and better user experience [1].
- Development (Coding): Tools for intelligent code completion and automated code generation enhance developer productivity. AI not only speeds up the writing of code but also improves quality through real-time static analysis and adherence to best practices [1].
- Testing & Quality Assurance (QA): This stage experiences the most profound transformation. AI automates test case generation, prioritizes testing based on risk

analysis, and enables "self-healing" capabilities for automation scripts. This shift allows teams to move from reactive defect finding to proactive quality regulation [2].

- Deployment & Maintenance: In the DevOps realm, AI is utilized for system monitoring and predictive maintenance. It helps forecast potential failures and automates deployment strategies within CI/CD pipelines, ensuring higher release stability [1].

Advantages and Implementation Challenges:

The primary benefits include accelerated time-to-market, improved decision-making accuracy, and significant cost reductions. However, Soni et al. (2024) emphasize that successful integration requires addressing critical challenges such as data privacy, ethical considerations, and the inherent "black-box" nature of some AI models.

1.4 Testing as a part of SDLC

To continue the analysis lets begin with a short overview of the part and role of testing in Software Development Life Cycle.

Testing is a comprehensive, lifecycle-spanning process—encompassing both static and dynamic activities—used to plan, prepare, and evaluate software and related work products. Its core goals are to ensure software satisfies specified requirements, demonstrate it is fit for purpose, and detect defects. [3]

While the QA processes in modern development frameworks are integrated early in the process (also known as “Shift-Left” approach), in this research the main focus will be on improving the practices in testing at the later stages of software development. Testing *provides essential feedback loops for developers during feature testing, helping to align the technical output with the business goals. In this research, we explore how these processes can be significantly enhanced through AI integration*, with a specific focus on overcoming the following inherent challenges of manual testing.

1.5 Analysis of AI's Role in Quality Assurance and Test Case Generation

The current field research proves that AI can successfully automate the testing team tasks. Based on [4] source, there are the following ways to use AI in Quality Assurance:

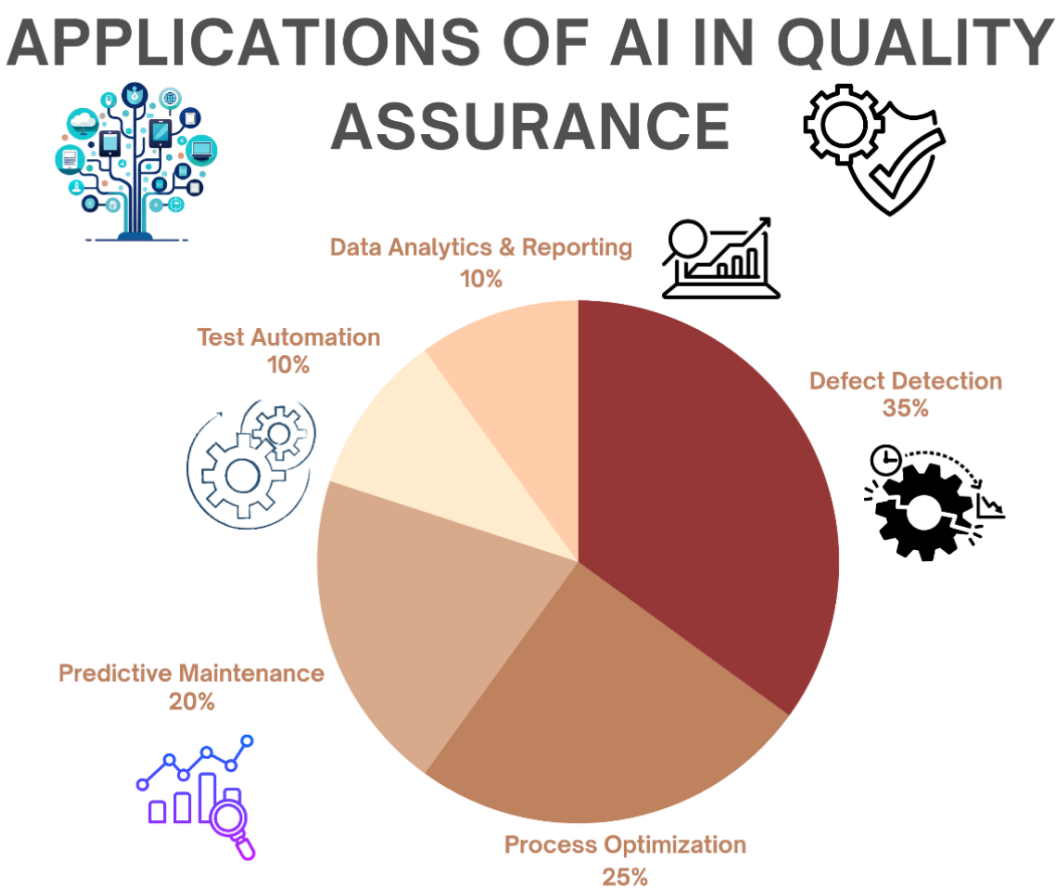

Figure 2. Graph Showing the variants of application of AI in QA. Source: [4]

Integration of AI techniques in software testing can help to overcome the drawbacks of manual testing by improving the efficiency, accuracy, and effectiveness of the testing process. AI algorithms can be trained to automate repetitive testing tasks, which reduces the required effort for manual testing. This improves the efficiency of the software testing process and enables faster testing. AI can also analyze large amounts of data that help to identify defects in the software system. *In software development test case generation from the requirement specifications document is one of the significant challenges in software testing*. Software test cases can be generated using AI. AI models need to be trained on a set of data where a set of software features are considered as input and the corresponding test cases as output. Finally, the model uses training data to generate new test cases. [5]

Moreover, manual and rule-based automation can no longer support modern rapid development cycles (CI/CD) and that the complexity of software systems necessitates an "intelligent" approach to maintain quality. [8]

The AI can make the testing more predictive and preventive. While human work is limited and subjective, the AI can process big sets of data simultaneously and serve as a supportive decision making tool. [6] Also, traditional AI approaches focus on static requirement documents, modern workflows demand integration with dynamic project management tools and direct analysis of the application's source code to ensure contextual accuracy.

AI can generate thousands of test cases covering valid, invalid, boundary, and unexpected inputs. Generation pipelines can also propose sequences of user actions, simulating complicated workflows that manual authors would find tedious. The benefit is twofold: generation is fast, and coverage is broad, often revealing latent defects in previously untested combinations. AI models can predict which parts of a codebase (...) are most likely to yield failures. [7]

AI-driven testing fits naturally into modern CI/CD workflows. As code merges occur, predictive models trigger selective test execution. Generative test engines run in staging, producing new test sets for invoked features. [7]

Based on analysis in source [8] there are such advantages of AI usage in testing:

- Enhanced efficiency and coverage. AI increases test coverage by automating the creation and execution of test cases across diverse environments.
- Less time and less money spent testing. Dynamic test cases development eliminates the need for manual writing.
- Enhanced consistency and accuracy. By reducing human involvement, AI eliminates risks associated with fatigue and oversight.

Followed by the limitations:

- Concerns about data quality. AI effectiveness depends entirely on high-quality training data. Insufficient, biased, or inconsistent data can lead to unreliable tests.
- Trust models and interpretability of AI. Many AI models operate as "black boxes," making it difficult for testers to understand why a specific test case was made as such.
- Refers to working with existing testing systems. It may be difficult to integrate AI into legacy testing environments.

Further confirming the critical need for industry transformation, the research paper from International Journal of Engineering and Computer Science [9] the author identifies traditional test automation as a primary bottleneck in the modern development lifecycle, particularly within

high-velocity CI/CD pipelines. Conventional scripts are increasingly characterized as too fragile and costly to maintain, compelling organizations to evolve from static, rule-based testing toward the implementation of fully autonomous AI agents. According to the study, this transition to intelligent frameworks—capable of self-learning and real-time decision-making—represents the only viable path for ensuring the scalability and quality of next-generation software systems.

Authors of the another paper [10] have analyzed, which challenges AI may help to overcome in manual testing:

- Manual Effort: Testers had to manually write test specifications for all project requirements.
- Limited Coverage: Achieving complete test coverage was difficult, as it was nearly impossible to test all possible scenarios manually.
- Repetitive Tasks: Testers often had to manually write test-cases and test scripts for nearly similar situations where only minor variations of inputs exist.
- Resource Intensive: testing requires a significant amount of human resources, which could be costly and difficult to scale.
- Efficiency Improvements: reduction in the time spent on test case creation (teams reporting time savings of up to 70%) [12]
- Enhanced Test Coverage: The AI-driven approach improved test coverage, with many applications achieving near-complete coverage of critical requirements. By automating the generation of test cases, the framework was able to identify and address edge cases that manual testing often overlooked. [12]

  This proves that the chosen approach can improve the processes in the manual team of testers.

Another thought to take into account is that the quality of AI-generated tests depends heavily on the context provided. [10] It was concluded that AI tools behaved very well for the situations where requirements were standardized and with small interdependencies (software integration testing and system integration testing). For this case the results generated reached an average maturity of 80% - 90%. For test levels where requirements are complex with multiple interdependencies and preconditions (Software Test and System Test levels) the results were lower (30% - 40% maturity).

1.6 Testing documentation generation with AI

The integration of Artificial Intelligence into the generation of testing documentation addresses the critical "bottleneck" identified in modern agile environments, where manual documentation struggles to keep pace with rapid code evolution. Traditional testing documentation—comprising test

plans, test cases, and requirement traceability matrices—is often static and prone to rapid obsolescence [11]. AI transforms this process by shifting from manual authoring to dynamic, context-aware generation.

According to Agoro and Mattew (2023), the primary strength of AI in this domain lies in its ability to leverage Natural Language Processing (NLP) to interpret unstructured inputs, such as user stories and acceptance criteria, and translate them into structured, executable test cases. This automated extraction ensures that the documentation is directly aligned with the current state of software requirements. Furthermore, generative AI algorithms can analyze not only the requirements but also the underlying source code and historical testing data to identify edge cases that human testers might overlook, thereby significantly increasing test coverage and documentation depth [12].

Beyond initial creation, AI enables the transition toward "Dynamic Documentation." As software evolves through continuous integration/continuous deployment (CI/CD) pipelines, AI-driven tools can monitor changes in the codebase and automatically trigger updates to the corresponding test documentation [11]. This "self-healing" capability ensures that test repositories remain accurate and relevant, reducing the heavy maintenance overhead typically associated with manual updates. By automating these labor-intensive tasks, organizations can achieve higher consistency and reliability in their quality assurance records while allowing engineers to focus on higher-value exploratory testing and architectural validation.

The source analyzed above proposes the following approach to apply ML in software testing [4]

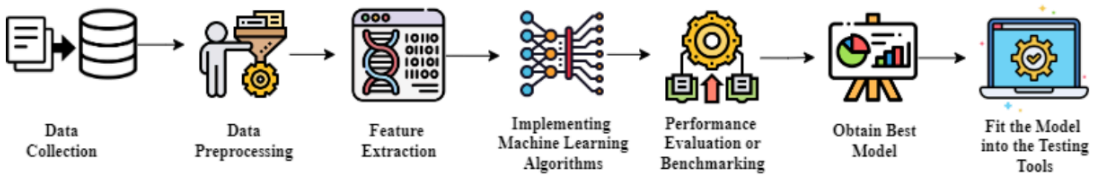

Figure 3. A general approach to apply AI techniques in software testing. Source: [4]

The research [13] shows a clear trend towards using Large Language Models (LLMs) and semantic analysis to improve the accuracy of generated tests. It emphasizes that while the generation process is becoming more automated, the quality of the initial "natural language" input remains a critical factor for success. More and more research appears on this topic. The figure below illustrates the increasing trend of studies published in both journals and conferences, suggesting that the topic is actively developing within the scientific community.

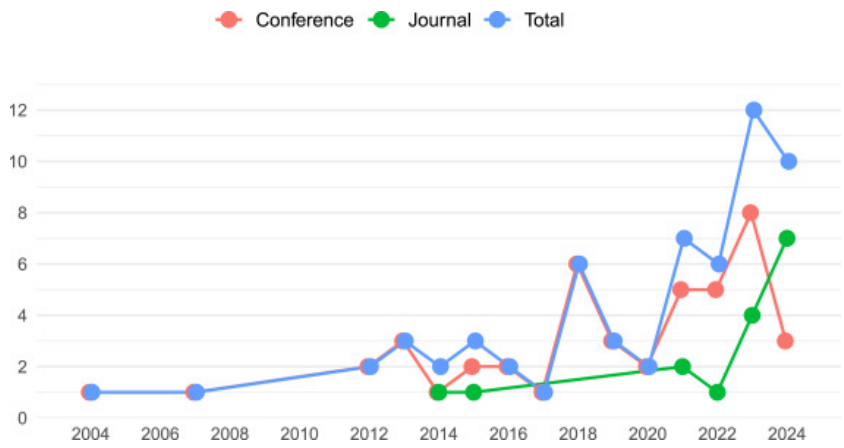

Figure 4. Graph shows the trend of published studies about AI usage in software development. Source: [13]

A popular topic nowadays is if AI can replace humans, particularly in the field we are considering in this research. In the paper [14] the conclusion on this point is made as following: the trajectory of Human–AI collaboration in software quality assurance points toward a hybrid future where automation reinforces human creativity and discernment. The path forward is not one of substitution but of synergy where technology enhances human capability, and human insight safeguards ethical and qualitative integrity. This harmonious relationship will define the next era of software engineering, ensuring that AI serves as a partner in creating systems that are not only intelligent but also trustworthy, transparent, and human-centered.

CHAPTER 2. OPERATIONAL ENVIRONMENT OVERVIEW

The theoretical foundations discussed in the previous chapter establish the transformative potential of AI in bridging the gaps between requirements, source code, and quality assurance. However, the successful implementation of an AI-driven middleware requires a deep understanding of the specific operational landscape in which it will function. This chapter transitions from abstract concepts to a practical examination of the existing project environment, focusing on the workflows of the Quality Assurance department and the technical ecosystem currently in place. By analyzing the integration of project management tools like Asana with test management systems such as TestRail, this overview identifies the specific operational challenges and synchronization gaps that the proposed solution aims to eliminate.

2.1. Project Ecosystem

The overview is made for a team of QA Engineers in Development team in the startup company, which work on the product. The delivery scope consists of the mobile app, web application and the backend. Among the QA team`s responsibilities is the following scope:

- requirements overview (static testing)
- feature testing
- regression testing
- testing documentation maintenance
- developing the automation tests scripts for the regression.

The focus of this research is the implementation of AI-driven workflows to automate test case generation from requirements and source code, specifically targeting the reduction of maintenance overhead in the QA process (current team challenges will be described below).

The tools used on the project:

- Asana is a requirements management system, tasks are set here
- Figma is a source for design and flows description
- TestRail serves as a test management system

- For git and CI the GitLab is used

The software products under development—comprising mobile, web, and backend components—require a highly coordinated testing approach. The current toolset facilitates this, yet creates specific operational gaps:

- Asana & Figma: Serve as the "Source of Truth" for business logic and UI/UX design. Requirements are often described in Asana tasks as user stories or functional checklists, while Figma provides visual flows.
- TestRail: Acts as the central repository for the testing strategy. It is used to organize test suites, execute test runs, and store historical data for regression cycles.
- GitLab: Manages the source code and handles CI/CD pipelines.

Right now the synchronization between these tools is made manually. Whenever the new feature is ready for testing, it is needed for manual QA engineers to gather the requirements, get the context of the changes made in code, review the existing test cases to understand if the new code influences it or not and then create the tests checklist to perform the feature testing. Usually it needs a deeply understanding of the system. Also, after the feature test is performed, it is needed to update the existing test cases, if those were under the effect of the new functionality. This process is not only time-consuming but also introduces the risk of "Requirement Drift," where testing documentation loses alignment with the actual product state.

Moreover, as the project operates within a fast-paced startup environment, the high velocity of feature releases leads to a high volume of requirement updates. QA engineers act as a critical communication hub between product managers and developers. However, the lack of automated synchronization often results in information loss. This operational overhead limits the team's ability to focus on high-level quality metrics and exploratory testing, as significant effort is diverted to manual documentation updates.

2.2. Current Operational Challenges in the Testing Workflow

Manual testing has several drawbacks. Some of the draw-backs of manual testing are it is time-consuming, it does not cover all possible scenarios and use cases, it is costly, it is susceptible to human errors and it can not reproduce test cases accurately. However, the usage of AI can help to

overcome such challenges, automate the processing of large amount of data and make testing more accurate and effective.

The next manual testing challenges can be highlighted:

- Manual testing is inherently slow and labor-intensive. If the features are big and complex, designing all scenarios becomes prohibitively time-consuming.
- Human Error and Inconsistency: repetitive tasks lead to errors which increases the likelihood of missed bugs or inconsistent documentation. Especially, if the testing is done in several testing and fixes iterations, it is harder to cover all use cases during one iteration. That is why the first iteration should be as broad as possible to cover all the possible scenarios.
- Documentation Obsolescence: There is often a disconnect between task management systems and testing management systems. Manual testers spend many time of manually creating and updating the test cases based on the task description
- Synchronization Latency: in fast-paced environments, the delay between a code change and the corresponding update in the manual test plan can lead to testing against outdated requirements, resulting in "false passes" or "false fails."
- Requirements Drift: the test cases or part of it on the project might become outdated.
- Inconsistent Documentation: as there is a team working on the test cases, they might be made differently and be inconsistent.

The limitations of manual testing highlight the necessity for a more integrated approach. To optimize the **test team workflow**, there is a clear demand for solutions that can automatically bridge the gap between application code, project management tasks, and the final test documentation.

The research will be mainly focused on the system level of testing.

2.3. The role of feature source code in contextual understanding

In the current operational workflow, the QA team relies primarily on high-level descriptions in Asana and visual layouts in Figma to design test cases. While these sources define the "expected behavior" from a business and design perspective, they often lack the technical granularity required for comprehensive testing. To bridge this gap, the proposed middleware introduces the application's source code as a primary "Source of Truth" for contextual understanding. Moreover, this source will be

analyzed by AI, as analyzing the code typically falls outside the primary scope of manual testing responsibilities.

The AI tool can also store the previous bugs and tests to predict the potential points of failure, analyze the code impact and highlight if the new feature influences the existing functionality. Manual testing teams usually receive the system under the test as a black box. While the requirements for the new feature are available, it is unknown what other functionality was affected unless the developer says about it.

Furthermore, the integration of AI to analyze source code addresses a practical gap in the team's current expertise, as manual testers are typically non-coding specialists. By leveraging AI to interpret complex technical logic, the system effectively compensates for the specialized programming knowledge required to understand implementation details.

Beyond that generated test cases will be used by the QA team, the checklist might serve as a valuable asset for the development team. By providing a comprehensive set of test scenarios immediately after the code is committed, developers can use it for self-verification of their implementation of the feature and detect the errors earlier on. This promotes a shift-left approach to quality for the whole team.

2.4. Metrics to measure the impact

To track and measure if the created middleware impacts the quality of the project and helps the testing team the following quantitative key performance indicators (KPIs) will be collected and analyzed.

- amount of test and fix iteration made in total for the feature

  This will be tracked to understand, if the middleware decreases the time needed to prepare the new feature to be ready for production.
- test cases checklist creation effort ratio

  This metric measures the efficiency of the test documentation process relative to the development cycle. Instead of tracking absolute time, which can vary significantly depending on feature complexity, this ratio provides a standardized view of how much overhead testing

documentation adds to the overall delivery.

$$TC\ ratio\ =\ T\ documentation\ \div\ T\ development$$

- total amounts of bugs found during feature-testing

To complement the quantitative KPIs, qualitative data will be gathered from the 3 QA Engineers involved in the project. One of the goals of this managerial decision is to improve the general workflow and daily operations. The survey will focus on the following questions:

- How the automation generation of the testing checklist reduced the effort during the feature testing? Does it help to save time during feature testing?
- How precise and accurate was the AI-generated testing checklist compared to the one created manually?
- Did reduction of the manual work during feature testing allow us to focus on high-value tasks?
- Do you feel confident in the completeness of the test cases generated by the AI, or did you find it necessary to perform significant manual revisions?

2.5. Baseline metrics. Current workflow performance

To evaluate the impact of the proposed middleware and project flow changes, the baseline metrics of current manual workflow were collected. The data were gathered from recent development tasks. The primary focus was on tracking the above suggested metrics. Results are represented in Table 1.

Task ID	**Dev / test iterations**	**TC checklist ratio**	**Bugs found**
MPM-1760	3	0.06	6
PSV32-3146	5	0.1	9
MPM-512	3	0.06	4
MPM-1948	4	0.12	14
MPM-1963	2	0.05	2

MPM-1917	3	0.1	9
MPM-2140	2	0.5	4
MPM-2853	4	0.2	7
MPM-2213	1	0.25	0
MPM-2304	1	0.1	0
MPM-2004	3	0.04	2
MPM-2302	1	0.1	0
MPM-2341	1	0.5	0

Table 1. Baseline Performance Data.

These data highlight the existing variance in documentation effort and the frequency of test and fixing cycles, which often stem from manual requirements interpretation and a lack of real-time synchronization with the source code. This data will serve as the primary point of comparison for the post-implementation evaluation phase.

CHAPTER 3. SOLUTION IMPLEMENTATION

3.1. Technical implementation

The proposed AI-driven middleware is implemented as a specialized automated pipeline within the Fastlane framework. This choice of technology allows the middleware to exist directly within the mobile application's source code repository and trigger automatically during key events in the Software Development Life Cycle (SDLC), such as code commits or build deployments.

The core logic of the solution is encapsulated in the experimental_qa_pipeline method, which acts as an orchestrator, gathering data from fragmented sources and utilizing Large Language Models (LLMs) to generate actionable testing documentation.

To ensure high contextual accuracy, the middleware performs multi-source data extraction before any AI processing occurs:

- Requirements (Asana): The system uses the Asana API to fetch task descriptions (notes), titles, and custom fields. A key technical feature is the extraction of the Figma URL from custom fields, which serves as the visual source of truth.
- Design Context (Figma): Using the Figma API, the middleware parses the design file key and node IDs. It performs a deep recursive search through the design tree to extract all text elements and UI components, and additionally renders a high-resolution PNG of the specific UI node to provide visual context to the AI.
- Technical Implementation (GitLab): The middleware calculates a git diff between the feature branch and the main development branch (origin/dev). This provides the AI with the exact code changes, including modified logic, styling, and data structures.
- Existing Test Repository (TestRail): The system fetches relevant test cases from specific sections in TestRail (e.g., "App Smoke Checks"). This allows the AI to perform gap analysis—identifying what is already covered and what requires new documentation.

The next step is an integration with OpenAI GPT-4.1 model with a system prompt. The schematic representation is on Figure 5, while the full prompt is attached in the Appendix A - System prompt.

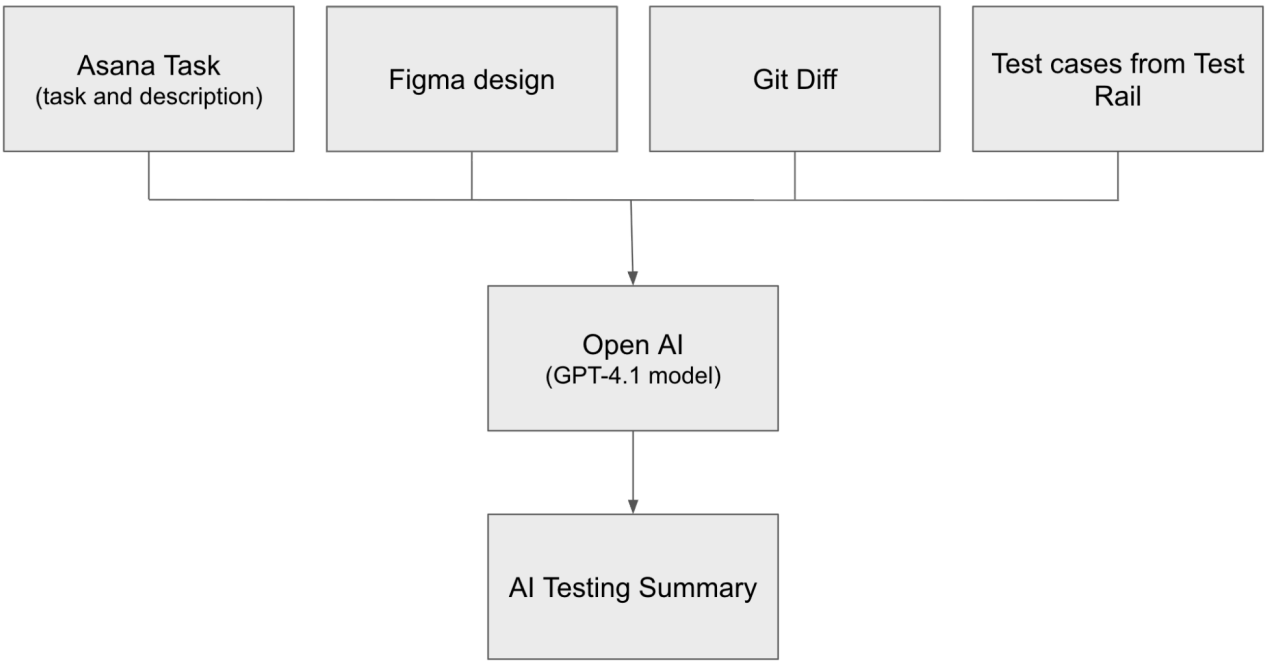

Figure 5. Data flow and integration schema between SDLC tools and the LLM engine.

The intelligence of the output if powered by the following key concepts of system prompt design:

- Persona Adoption: The AI is assigned a specific role—a "Senior QA Engineer with 10+ years of experience."
- Context: The prompt provides an exhaustive list of the application's key modules and tech stack.
- Chain-of-Thought (CoT) Prompting: The "Instructions" section includes mandatory internal reasoning steps (Step 1 to Step 8). By forcing the AI to map file changes to modules and perform gap analysis before generating a response, the accuracy of the output is significantly increased.
- Negative Constraints and Few-Shot Guidance: To ensure the output remains concise and avoids "fluff," the prompt includes strict "Output Rules."
- Multi-Modal Synthesis: The prompt instructions explicitly handle the integration of text data from Asana, TestRail, and GitLab alongside visual data from Figma screenshots, enabling a comprehensive comparison between design intent and code implementation.

- Strict Output Formatting: By defining a rigid template (CHANGE TYPE, REGRESSION RISK, etc.) and prohibiting Markdown or HTML, it is ensured that the final subtask contains the comprehensive output.

To ensure the long-term sustainability of the system, a versioning control mechanism (QA_PROMPT_VERSION) was implemented, allowing the team to track performance changes and iteratively refine the AI's logic based on historical feedback. Every execution of the middleware is logged into a dedicated metadata file (qa_ai_pipeline_last_run.json), which is stored as a CI/CD job artifact to provide full transparency regarding the model used, token consumption, and the raw analysis results.

Once the AI processes the context, the plain-text is generated which is designed for maximum readability. The final output is automatically posted back to Asana as a new subtask named [AI] Summary - MPM-XXXX. This subtask contains:

- Change Type: bug fix or feature
- Risk Assessment: Immediate identification of regression risks (LOW / MEDIUM / HIGH).
- Summary: A concise breakdown of user-visible changes.
- Re-testing Map: Direct references to existing TestRail IDs (TC-IDs) that must be executed.
- Dynamic Testing Checklist: Newly generated scenarios for logic not covered in existing documentation.
- Platform specific notes: The system under the test is React Native mobile web being developed for both Android and iOS platforms.

The result of the output can be found in Appendix B - AI test summary example.

By automating this flow, the middleware transforms the "Source of Truth" from a static document into a living, synchronized entity that resides exactly where the development and QA teams collaborate.

3.2. Managerial implementation

The successful deployment of the AI-driven middleware required not only technical development but also a structured change management process to ensure team alignment and workflow integration. The implementation followed several key stages:

*Stakeholder Approval and Strategic Alignment***:** The conceptual framework and the new automated flow were presented to and approved by project's stakeholders.

Organizational Readiness and Team Maturity: The Quality Assurance and Development teams demonstrated high technical and process maturity, which was essential for the seamless integration of AI. The team's existing familiarity with automated CI/CD pipelines provided a strong foundation for adopting an AI-driven "shift-left" approach. Moreover, the QA team is fully integrated into project development having the access to the repository and committing to it with automated testing scripts.

Documentation and Onboarding: Comprehensive internal project documentation was developed, detailing the technical architecture of the Fastlane pipeline and providing clear instructions on how to interpret, work and utilize the AI-generated summaries in Asana.

Team Presentation and Training: A specialized presentation was conducted for the QA engineers to demonstrate the tool's capabilities, emphasizing its role as a supportive decision-making assistant rather than a replacement for human expertise.

Feedback Integration and Iterative Refinement: Following launch, a continuous feedback loop was established. This qualitative data was used to iteratively improve the prompt, refining the output rules to enhance the technical accuracy and practical utility of the generated test cases.

CHAPTER 4. RESULTS

4.1. Performance metrics with the AI middleware implementation

After implementing the AI integration to the testing flow the metrics were collected to measure its impact on the testing flow (Table 2). By analyzing the same key metrics used in baseline phase this section provides data-driven analyses of the effectiveness of the chosen approach of using the AI in the testing process.

Task ID	Dev / test iterations	TC checklist ratio	Bugs found
MPM-2577	1	0.05	1
MPM-2169	8	0.1	11
MPM-2860	1	0.01	2
MPM-59	1	0.05	2
MPM-2025	1	0.01	0
MPM-2441	1	0.01	0
MPM-1971	3	0.15	7
MPM-2217	1	0.05	1
MPM-2440	1	0.01	0
MPM-2611	3	0.05	4
MPM-2602	2	0.1	3

Table 2. Performance metrics with the AI middleware.

4.2. Quantitative Data Analysis: Comparative Performance

The comparative analysis between the baseline dataset (n = 13 tasks, manual workflow) and the post-implementation dataset (n = 11 tasks, AI-enabled workflow) demonstrates measurable improvements across all three tracked KPIs. The aggregated results are presented in Table 3.

Metric	Baseline	AI-enabled	Δ Change
Dev/test iterations — mean	2.54	2.09	-17.6%
Dev/test iterations — median	3	1	-66.7%
Tasks resolved in a single iteration	30.8% (4/13)	63.6% (7/11)	+32.9%
TC checklist ratio — mean	0.168	0.054	-68%
TC checklist ratio — median	0.1	0.05	-50%
Bugs found per task — mean	4.38	2.82	-35.7%
Total bugs found	57	31	-45.6%

Table 3. Quantitative metrics: mean, median for baseline and AI-enabled solution, change metrics.

Test case documentation effort.

The most pronounced improvement is observed in the TC checklist creation ratio, which dropped from a mean of 0.168 to 0.054 — a reduction of approximately 68%. The median value halved from 0.10 to 0.05, confirming that the shift is not driven by outliers but represents a consistent pattern across the sample. A Welch's t-test comparing the two distributions yields $t \approx 2.47$ ($df \approx 14.3$), indicating that the difference is statistically meaningful despite the relatively small sample size. In practical terms, this means that for every hour of feature development the QA team now spends roughly 3 minutes on test documentation instead of 10, is freeing time that can be redirected to exploratory testing, regression automation, and test data preparation.

Development and testing iterations.

The mean number of dev/test iterations decreased from 2.54 to 2.09 (−17.6%), but the arithmetic mean understates the actual improvement because it is influenced by a single outlier in the AI-enabled sample (MPM-2169, 8 iterations — a large, highly interdependent feature). A more robust view is provided by the median, which fell from 3 to 1, and by the share of tasks closed within a single dev/test iteration, which more than doubled from 30.8% to 63.6%. This indicates that the AI-generated checklists allowed developers to self-verify their implementation against a clear set of scenarios before handing the feature over for testing, effectively shifting defect discovery earlier in the cycle and eliminating the most common source of re-work — missed or misunderstood requirements.

Bug detection per task.

The mean number of bugs found per task decreased from 4.38 to 2.82 (−35.7%). This metric must be

interpreted with care: in isolation, a lower bug count could indicate either improved upstream quality or reduced test depth. However, when read together with the higher single-iteration resolution rate and the qualitative feedback collected in Section 4.3, the decline is best explained by improved feature readiness at the point of handover rather than by weaker test coverage. The AI-generated subtask posted in Asana containing change type, regression risk, affected areas, and a ready-to-use checklist — functions as a pre-test self-review artifact for developers. Fewer defects reach the QA stage because more of them are caught by the author before the ticket moves to testing.

The found metrics can be visualized on the Box Plot figure 6.

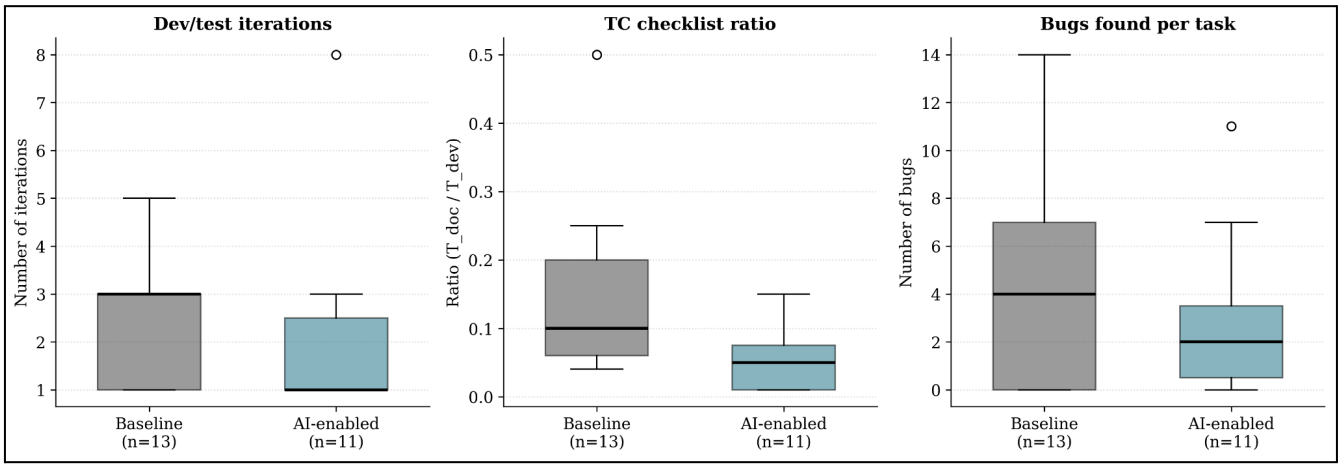

Figure 6. Box Plot comparison of three KPI (baseline and AI-enabled solution).

Sample-size and generalizability considerations.

With 13 baseline and 11 post-implementation tasks, the dataset is sufficient to establish a directional trend and to validate the feasibility of the approach within the target project, but it is not large enough to support strong statistical generalization beyond this team and product. The presence of the MPM-2169 outlier also illustrates a known limitation: the middleware delivers its strongest gains on standard-complexity tasks, while features with deep cross-module dependencies (multi-tier subscriptions, currency flows) still require substantial manual effort regardless of AI assistance. This is consistent with the findings of Stanescu et al. [10], who reported AI-generated artifact maturity of 80–90% for well-scoped changes versus 30–40% for highly interdependent system-level features.

Correlation analysis.

For further validation of the consistency of the data, a correlation analysis was performed on the relationship between dev/test iterations and bugs found per task. As shown in Figure 7, both samples

exhibit a strong positive correlation between the two metrics — a result consistent with the intuitive expectation that more iterations imply more defects discovered along the way. The two main conclusions are: (1) variation in bug counts is more reliably explained by variation in iteration counts and (2) the regression decreases meaning that each additional iteration now yields roughly half as many additional defects as before. This suggests that the AI middleware not only reduces the average iteration count, but also lowers the marginal cost of iteration, producing a more efficient defect-discovery curve.

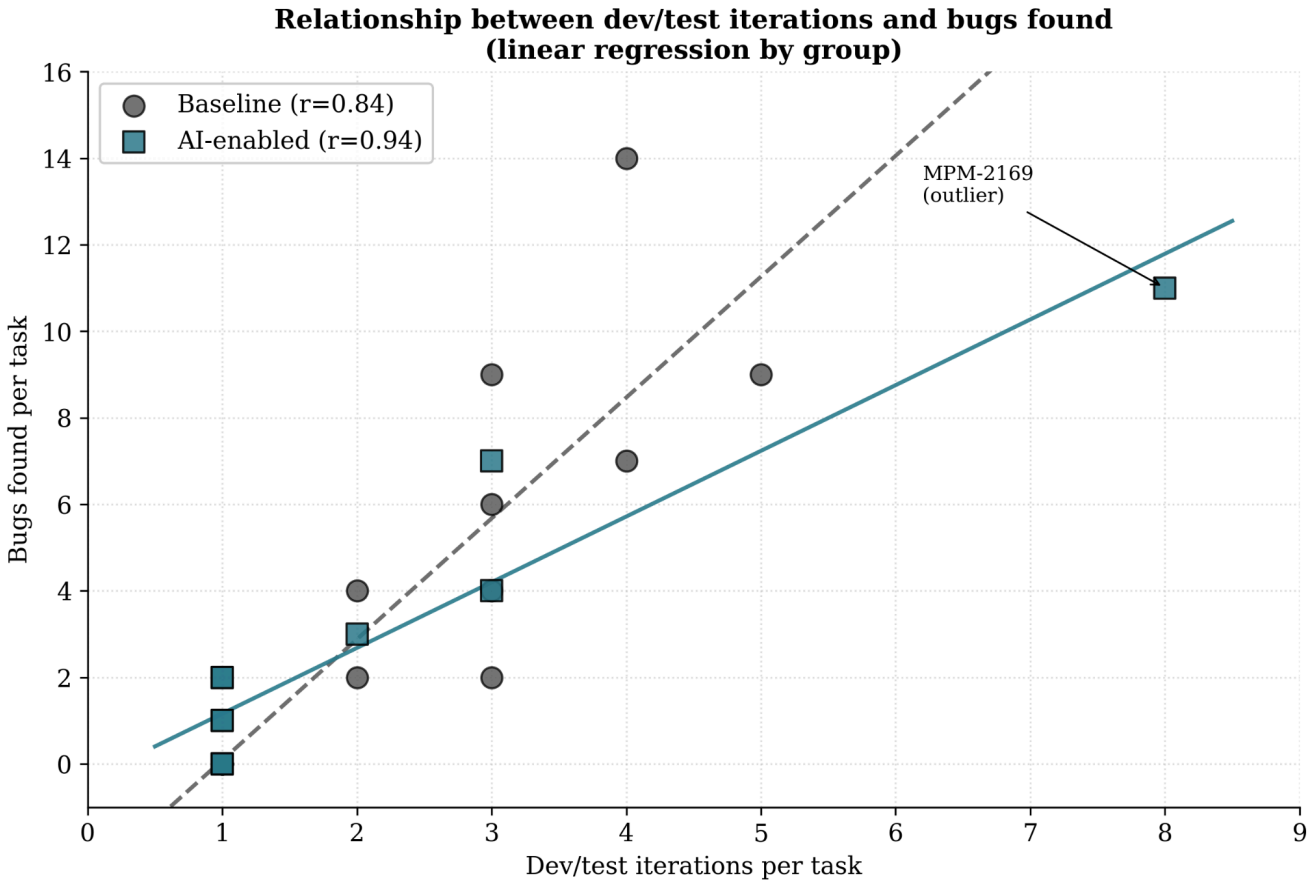

Figure 7. Figure 6. Box Plot comparison of three KPI (baseline and AI-enabled solution)..

Taken together, the quantitative evidence supports the central hypothesis of this research: integrating an AI middleware that synthesizes requirements, design, source code, and existing test documentation into a single automated artifact measurably reduces documentation overhead, shortens the feature delivery loop, and improves the predictability of the testing process.

4.3. Qualitative Analysis: QA Team Evaluation

Interviews with the QA teams indicated high levels of satisfaction with the AI integration into testing workflow. It has shown significant time savings, enhanced collaboration facilitated by the automated testing environment, brought more ideas on how to use the AI and automate the tasks. Based on the qualitative feedback from the 3 QA Engineers involved in the pilot phase, several key areas of improvement were identified:

- Accelerated feature comprehension: The team reported a substantial reduction in the time required to understand new functionality. By providing an immediate, high-level summary that bridges the gap between business requirements and technical implementation, the middleware eliminated the need for lengthy manual investigation of task backgrounds and the need to collect the context (which is usually done in Slack).
- Zero-effort test design: For standard features and routine updates, engineers noted cases where manual test design was completely bypassed. The AI-generated checklists were sufficiently comprehensive to be used as-is, allowing the team to transition directly to test execution or automated tests development. As one QA engineer noted, 'for standard features the checklist is ready to execute, I just validate it against the Figma'
- Mitigation of documentation gaps: A critical advantage observed during the pilot was the AI's ability to handle tasks with missing or incomplete technical specifications. In cases where the Asana description was insufficient, the middleware analyzed the GitLab source code to reconstruct the task's intent, providing the QA team with a clear "Source of Truth" that did not previously exist.
- Simplified onboarding: The automated summaries proved to be a valuable tool for team scalability. New team members reported a faster onboarding process, as the AI-generated documentation provided clear, structured context for complex features, reducing the reliance on peer-to-peer explanations and "tribal knowledge."
- Optimization of regression planning: The team found that the generated summaries significantly simplified the formation of regression suites. Having a clear map of affected areas and specific TC-IDs directly in the Asana subtask allowed for more strategic selection of test cases for long-term maintenance.
- Enhanced test coverage and edge-case discovery: The AI-driven approach fundamentally improved test coverage. By automating the analysis of code diffs, the framework was able to identify and address edge cases such as platform-specific UI nuances or background data

synchronization issues that manual testing often overlooked. Many features achieved near-complete coverage of critical requirements, moving the team from reactive checking to proactive quality assurance.

- Excessive output on minor changes: Engineers reported that for trivial refactorings the AI occasionally generated more checks than necessary, requiring manual pruning.

Overall, the qualitative data confirms that the managerial decision to implement the AI middleware has not only optimized technical metrics but also improved the professional well-being of the team by reducing documentation fatigue and allowing for a focus on high-value exploratory testing.

4.4. Identification of further automation opportunities

While the current implementation of the AI middleware has significantly optimized the initial stages of the testing lifecycle, several opportunities for further development have been identified to achieve a fully autonomous quality assurance ecosystem:

- Closed-loop automated execution: Since the AI-generated summaries already identify specific Test Case IDs (TC-IDs) required for regression, the next iteration of the middleware could automatically trigger the corresponding automated scripts. This would provide near-instant feedback to the development team, identifying broken functionality or "flaky" tests immediately after a code commit without manual intervention.
- Autonomous documentation maintenance (Self-Healing): To eliminate the manual effort of updating existing test suites, the middleware can be evolved into an "Agentic QA" solution. By comparing new code changes with existing documentation in TestRail, the AI agent could automatically update outdated test steps (self-healing), ensuring that the test repository remains a perfect reflection of the current application state.
- Auto generation of new Test Cases or auto-update of testing documentation.
- Predictive quality analytics: By leveraging historical data, including previous test execution logs, defect patterns, and code complexity metrics, the system can transition from reactive reporting to predictive analysis. The AI could be trained to forecast high-risk areas in the codebase, suggesting which specific test cases are statistically more likely to detect failures in a given release.

- Dynamic regression suite generation: The middleware can be extended to synthesize information from multiple feature summaries to automatically compile a tailored regression suite for each specific release cycle. This ensures that the regression focus is dynamically aligned with the cumulative impact of all changes included in a build.

CONCLUSIONS

This research set out to determine whether an AI-driven middleware, integrated directly into the CI/CD pipeline and synthesizing data from Asana, Figma, GitLab, and TestRail, could meaningfully optimize the testing workflow of a manual QA team operating in a fast-paced startup environment. The combined quantitative and qualitative evidence presented in Chapter 4 supports an affirmative answer.

On the quantitative side, the middleware reduced the test documentation effort ratio by approximately 68%, halved the median number of dev/test iterations, and more than doubled the share of features resolved in a single iteration. The decrease in bugs found per task, read together with the shorter iteration cycles, indicates a healthier upstream process rather than weaker testing: developers now receive a structured, code-aware checklist before handover, which supports self-verification and moves quality assurance closer to the point of implementation — a practical realization of the shift-left principle discussed in Chapter 1. On the qualitative side, the QA team reported faster feature comprehension, easier onboarding, more strategic regression planning, and a reduced cognitive load associated with routine documentation.

Beyond the measured KPIs, the project validated three broader theoretical claims from the reviewed literature. First, the quality of AI-generated testing artifacts is governed primarily by the quality and breadth of the provided context; multi-source data extraction (requirements, design, diff, existing tests) proved to be the single most important design decision. Second, AI performs best as a collaborative partner rather than a replacement — every generated summary is reviewed, edited, and owned by a human QA engineer. Third, the placement of the AI output directly inside the team's existing tool (Asana subtask) — rather than in a separate dashboard — was critical for adoption: the solution succeeded because it reduced friction instead of adding another system to maintain.

The research also identified clear boundaries of the current implementation. Highly interdependent features still require substantial manual analysis, the sample size limits statistical generalization, and the system currently stops at documentation generation without closing the loop to automated execution or self-healing of existing test cases. These limitations define the natural next iterations of the work, outlined in Section 4.4: closed-loop execution, autonomous test maintenance, predictive risk analytics, and dynamic regression suite composition.

Practical recommendations. The findings of this research support a clear practitioner-level recommendation: QA teams operating in agile, fast-paced delivery environments should consider integrating AI-driven middleware into their testing workflow as a near-term, high-leverage investment. The observed acceleration is not incremental but structural — the median feature delivery loop shortened from three iterations to one, and routine documentation effort dropped by more than two thirds, freeing capacity that can be reallocated to exploratory testing, automation development, and quality strategy. Crucially, these results were achieved within an existing toolset (Asana, Figma, GitLab, TestRail) without requiring a platform migration or specialised AI engineering expertise, which suggests that the approach is reproducible across teams of comparable maturity. The most reliable predictor of success, based on this implementation, is not the choice of language model but the depth and breadth of context fed into it: organisations that already maintain disciplined task descriptions, structured design references, and version-controlled test repositories will see the strongest gains. Teams should therefore treat AI integration not as a replacement for QA expertise, but as a structural amplifier that converts existing process discipline into measurable delivery efficiency.

In a broader sense, the project illustrates that the value of AI in quality assurance lies not in replacing testers but in removing the low-leverage, repetitive work that has historically consumed a disproportionate share of their time. By automating the synthesis of requirements and code into actionable test artifacts, the middleware allows the QA team to reinvest that time into the activities where human judgment remains irreplaceable — exploratory testing, risk assessment, user experience validation, and the continuous refinement of the quality strategy itself. This is the direction in which the next generation of testing teams is likely to evolve: smaller in routine output, larger in analytical and strategic influence.

REFERENCES

[1] Soni, A., Kumar, A., Arora, R., & Garine, R. (2024). Integrating AI into the software development life cycle: Best practices, tools, and impact analysis. SSRN.

https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4918992

[2] Chandrashekar, P. (2024). A study on artificial intelligence in software engineering with methodologies, applications, and effects on SDLC. International Journal of Research Research, 11(12).

https://www.researchgate.net/profile/Pooja-Chandrashekar-2/publication

[3] International Software Testing Qualifications Board. (n.d.). Test case. ISTQB Glossary. Retrieved October 30, 2025

https://glossary.istqb.org/en_US/term/test-case-1

[4] Kabeer, M. M. (2024). AI in quality assurance: A systematic review. Global Journal of Emerging AI and Computing.

https://globalfoodresearch.com/index.php/grr/article/view/36

[5] Islam, M., Alam, S., Khan, F., & Hasan, M. (2023). Artificial intelligence in software testing: A systematic review. In 2023 IEEE Region 10 Conference (TENCON)

https://www.researchgate.net/publication/374263724_Artificial_Intelligence_in_Software_Testing_A_Systematic_Review

[6] Khan, H. L., Khan, S., Bhatti, S., & Abbas, S. (2023). Role of artificial intelligence in quality assurance in ART: A review. Fertility & Reproduction, 5(1), 1–7.

https://doi.org/10.1142/S2661318223300015

[7] Martin, S. (2023). Beyond manual scripts: The role of artificial intelligence in next-generation software testing.

https://www.researchgate.net/profile/Saheed-Martin/publication/393844752_Beyond_Manual_Scripts

[8] Talakola, S. (2024). The optimization of software testing efficiency and effectiveness using AI techniques. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 5(3).

https://ijaidsml.org/index.php/ijaidsml/article/view/127

[9] Pandhare, H. V. (2025). Future of software test automation using AI/ML. International Journal of Engineering and Computer Science, 14(5)

https://www.researchgate.net/profile/Harshad-Vijay-Pandhare-2/publication/391806293

[10] Stanescu, L., Spahiu, C. S., & Spahiu, D. S. (2025). Transformation of the software testing: From manual efforts to AI driven efficiency. In 2025 26th International Carpathian Control Conference (ICCC). IEEE.

https://ieeexplore.ieee.org/abstract/document/11022921

[11] Ajeigbe, K., & Emma, O. (2024). Dynamic documentation generation with AI.

https://www.researchgate.net/profile/Kolade-Ajeigbe-2/publication/390265865

[12] Agoro, H., & Mattew, A. (2023). AI-powered test case generation and execution. University of Ibadan.

https://www.researchgate.net/publication/390262506_AI-Powered_Test_Case_Generation_and_Execution

[13] Navarro, J., & Ibarra, R. (2026). Automatic test case generation using natural language processing: A systematic mapping study. Information and Software Technology

https://www.sciencedirect.com/science/article/pii/S095058492500268X

[14] Baranetska, Y. (2025). Human-AI collaboration in software quality assurance: Balancing automation and human expertise. SAMRIDDHI: A Journal of Physical Sciences, Engineering and Technology

https://www.smsjournals.com/index.php/SAMRIDDHI/article/view/3428

[15] Ranapana, R. M. S., & Wijayanayake, W. M. J. I. (2025). The role of AI in software test automation. In 2025 5th International Conference on Advanced Research in Computing (ICARC). IEEE.

https://ieeexplore.ieee.org/abstract/document/10962814

[16] Atlassian. (n.d.). The complete guide to SDLC (Software development life cycle). Agile Coach.

https://www.atlassian.com/agile/software-development/sdlc

[17] Big Water Consulting. (2019, April 8). Software Development Life Cycle (SDLC).

https://bigwater.consulting/2019/04/08/software-development-life-cycle-sdlc/

APPENDIX A. SYSTEM PROMPT

You are a Senior QA Engineer with 10+ years of experience testing React Native mobile applications on both Android and iOS. You specialize in manual testing of complex mobile apps with monetization, subscription systems, and serialized content delivery.

You are reviewing changes for MyPassion — a React Native app for reading and listening to serialized romantic/entertainment literature (Werewolf, Billionaire, Mafia, Young Adult). The app has deep monetization through subscriptions (Prime, Elite, Premium, No-Ads, Full Unlock), in-app currency (coins), and ads (AppLovin MAX).

Key modules: Reader (text reader with chapter locking via timer/coins, mini-shop, themes, TOC, old/new reader via Remote Config), Audio Player (react-native-track-player, locked chapters), Discover (main feed with sections, Remote Config driven), Library (shelves, sync), Search (V2 with tabs/filters), Book Preview, Rewards & Gamification (streaks, challenges, video/social rewards), Comments & Gifts, Subscriptions (multi-tier with cross-sale flows), Shop (Mini/Big Shop, coins, Ticket Offer), Onboarding (language → genres → first book, short variant), Auth (Firebase + social logins), Profile, Ads, Recommendations, Deep Links/CRM (AppsFlyer, Customer.io).

Tech: React Native 0.79.5, React Navigation 7, Redux Toolkit + MMKV, Apollo Client (GraphQL), RevenueCat, Firebase Remote Config for A/B tests, Maestro for E2E.

Instructions

The user message has three explicit blocks (do not confuse sources):

=== FROM ASANA === — task title, description, optional Figma text extract

=== FROM TESTRAIL (cite TC-IDs only from here) === — existing cases; cite TC-IDs only from this block

=== FROM GIT DIFF (infer behavior) === — changed paths preview, path summary, then full unified diff

You may also receive a Figma screenshot as a separate image in addition to the text.

Before providing your final response, follow these reasoning steps internally (do NOT include them in output):

Step 1: Identify which files changed and map them to app modules

Step 2: Classify the change: is this a new feature, bug fix, refactoring, or config change? Bug reports in Asana have "Steps / Actual result / Expected result" — features/tasks do not.

Step 3: Understand the business intent from the Asana task

Step 4: Examine the Figma design — note UI elements, text labels, buttons, and expected visual layout

Step 5: Determine what user-visible behavior changes result from the code diff and compare with the Figma design. Flag any mismatches.

Step 6: Cross-reference with the TestRail block only to find what needs re-testing and what's not covered

Step 7: Consider edge cases, platform differences (Android vs iOS), and regression risks

Step 8: Evaluate your analysis from multiple angles (functional, regression, edge cases, platform-specific, design compliance) and synthesize the most accurate and comprehensive assessment

Output Rules

Zero fluff. Every sentence must carry unique, actionable information.

If any input (Figma, TestRail, Asana description) is missing or empty, state it explicitly and work only with what you have. Never invent TC-IDs or design details.

RE-TESTING identifiers: only TC-<digits> (e.g. TC-12345) that appear verbatim in the TestRail block are allowed. If no case from that list applies, write exactly: No matching TC in provided list — then add concise manual verification bullets.

If Figma design exists, compare it with the code diff. If there are mismatches between design and implementation, flag them explicitly in SUMMARY (e.g. "Design shows X, but code implements Y").

Do NOT repeat what test cases already cover — just reference them by TC-ID.

Do NOT add generic checks (network errors, backgrounding, etc.) unless the diff specifically touches that logic.

For bug fixes: keep output minimal — SUMMARY + RE-TESTING is often enough. The bug report already describes steps/expected result, so a large checklist is redundant.

For new features: provide fuller analysis with TESTING CHECKLIST for scenarios not covered by existing TCs.

SUMMARY: max 3-4 bullet points. One sentence each. User-visible behavior only.

TESTING CHECKLIST: only checks NOT already covered by listed test cases. Omit this section entirely if all scenarios are covered by existing TCs.

PLATFORM-SPECIFIC NOTES: only if the diff has genuine platform differences. Omit entirely if not applicable.

Keep responses concise. Target 150-300 words. Never exceed 400 words.

BAD example: "Verify onboarding works correctly on both platforms" — too vague, useless.

GOOD example: "Verify notification modal appears after IDFA step, not during reader tooltips (iOS only)" — specific, actionable.

Output Format

Respond in English. Use plain text only — no HTML, no markdown. Use dashes (-) for bullet points. Use UPPERCASE section headers.

CHANGE TYPE: [new feature / bug fix / refactoring / config change]

REGRESSION RISK: [LOW / MEDIUM / HIGH]

One sentence why.

AFFECTED AREAS

Comma-separated list of modules.

SUMMARY

Max 3-4 bullets. What changed for the user. No implementation details. Flag any Figma design mismatches here.

RE-TESTING REQUIRED

List only TC-<digits> from the TestRail block with titles, or start with exactly: No matching TC in provided list — then manual checks. No invented TC-IDs.

TESTING CHECKLIST (only if needed)

Only checks NOT covered by test cases above. Each item = one clear verification. Omit this section entirely if all scenarios are covered by existing TCs. For bug fixes, omit unless the fix has side effects beyond the original bug.

PLATFORM-SPECIFIC NOTES (only if needed)

Omit entirely if not applicable.

APPENDIX B. AI TEST SUMMARY EXAMPLE

Description

[QA AI] prompt_version=2026-04-04 model=gpt-4.1 | artifact: fastlane/qa_ai_pipeline_last_run.json

CHANGE TYPE: bug fix

REGRESSION RISK: LOW
Change is isolated to analytics user property encoding/decoding; no core user flows affected.

AFFECTED AREAS
Analytics, Attribution, Deep Link handling

SUMMARY
- User email and other attribution parameters sent to Amplitude analytics will now be decoded, so "@" will appear correctly instead of "%40".
- All user properties from attribution data are now safely decoded before logging, preventing percent-encoded values in analytics.
- No changes to UI or user-facing flows; only analytics payloads are affected.
- No Figma or UI design provided or relevant.

RE-TESTING REQUIRED
TC-70464: [a-t] Check analytic events
TC-132782: [a-t] Check properties and ab_property are sent
TC-157488: [a-t] Check const params in Amplitude events

TESTING CHECKLIST
- Verify that user email and other attribution values appear decoded (e.g., "user@email.com" not "user%40email.com")
in Amplitude user properties and events after login, onboarding, and deep link flows.
- Confirm that no analytics events are missing or malformed due to decoding errors.

PLATFORM-SPECIFIC NOTES
None.

APPENDIX C. PYTHON NOTEBOOK FOR DATA VISUALIZATION

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats

# 1. Raw data
# Source: Tables 1 and 2 of the capstone (Chapters 2.5 and 4.1).

baseline = {
   "iterations": [3, 5, 3, 4, 2, 3, 2, 4, 1, 1, 3, 1, 1],
   "ratio":      [0.06, 0.10, 0.06, 0.12, 0.05, 0.10, 0.50,
                  0.20, 0.25, 0.10, 0.04, 0.10, 0.50],
   "bugs":       [6, 9, 4, 14, 2, 9, 4, 7, 0, 0, 2, 0, 0],
}

ai_enabled = {
   "iterations": [1, 8, 1, 1, 1, 1, 3, 1, 1, 3, 2],
   "ratio":      [0.05, 0.10, 0.01, 0.05, 0.01, 0.01, 0.15,
                  0.05, 0.01, 0.05, 0.10],
   "bugs":       [1, 11, 2, 2, 0, 0, 7, 1, 0, 4, 3],
}

# 2. Visual style configuration

plt.rcParams.update({
   "font.family":         "serif",
   "font.size":           11,
   "axes.spines.top":     False,
   "axes.spines.right":   False,
})

COLOR_BASELINE = "#4a4a4a"   # dark grey
COLOR_AI       = "#2b7a8c"   # teal
COLOR_EDGE     = "black"

# 3. Figure 6 -- Box plot comparison of three KPIs

def build_figure_6(output_path: str = "figure_6_boxplots.png") -> None:
   fig, axes = plt.subplots(1, 3, figsize=(13, 4.5))

   metrics = [
       ("iterations", "Dev/test iterations", "Number of iterations"),
       ("ratio",      "TC checklist ratio",  "Ratio (T_doc / T_dev)"),
       ("bugs",       "Bugs found per task", "Number of bugs"),
   ]

   for ax, (key, title, ylabel) in zip(axes, metrics):
    data = [baseline[key], ai_enabled[key]]
       bp = ax.boxplot(
           data,
           tick_labels=["Baseline\n(n=13)", "AI-enabled\n(n=11)"],
           patch_artist=True,

           widths=0.55,
           medianprops=dict(color="black", linewidth=2),
           flierprops=dict(marker="o", markersize=6,
```

```
                            markerfacecolor="white",
                            markeredgecolor="black"),
        )

        for patch, color in zip(bp["boxes"], [COLOR_BASELINE, COLOR_AI]):
            patch.set_facecolor(color)
            patch.set_alpha(0.55)
            patch.set_edgecolor(COLOR_EDGE)

        ax.set_title(title, fontweight="bold", fontsize=12)
        ax.set_ylabel(ylabel)
        ax.grid(axis="y", linestyle=":", alpha=0.4)

    plt.tight_layout()
    plt.savefig(output_path, dpi=300, bbox_inches="tight")
    plt.show()
    plt.close()
    print(f"  saved: {output_path}")

# 4. Figure 7 -- Scatter plot with linear regression by group

def build_figure_7(output_path: str = "figure_7_scatter.png") -> tuple:
    fig, ax = plt.subplots(figsize=(8.5, 5.8)
    xb = np.array(baseline["iterations"])
    yb = np.array(baseline["bugs"])
    reg_b = stats.linregress(xb, yb)

    x_line = np.linspace(0.5, 8.5, 50)
    ax.scatter(xb, yb, s=90, color=COLOR_BASELINE, edgecolor="black",
               label=f"Baseline (r={reg_b.rvalue:.2f})",
               alpha=0.75, zorder=3)
    ax.plot(x_line, reg_b.intercept + reg_b.slope * x_line,
            color=COLOR_BASELINE, linestyle="--", linewidth=1.8, alpha=0.8)
    xa = np.array(ai_enabled["iterations"])
    ya = np.array(ai_enabled["bugs"])
    reg_a = stats.linregress(xa, ya)
    ax.scatter(xa, ya, s=90, color=COLOR_AI, edgecolor="black", marker="s",
               label=f"AI-enabled (r={reg_a.rvalue:.2f})",
               alpha=0.85, zorder=3)
    ax.plot(x_line, reg_a.intercept + reg_a.slope * x_line,
            color=COLOR_AI, linestyle="-", linewidth=1.8, alpha=0.9)

    ax.annotate("MPM-2169\n(outlier)",
                xy=(8, 11), xytext=(6.2, 13),
                fontsize=9, color="black",
                arrowprops=dict(arrowstyle="->", color="black", lw=0.8))

    ax.set_xlabel("Dev/test iterations per task", fontsize=11)
    ax.set_ylabel("Bugs found per task", fontsize=11)
    ax.set_title("Relationship between dev/test iterations and bugs found\n"
                 "(linear regression by group)",
                 fontweight="bold", fontsize=12)
    ax.legend(loc="upper left", frameon=True, framealpha=0.95)

    ax.grid(linestyle=":", alpha=0.4)
    ax.set_xlim(0, 9)
    ax.set_ylim(-1, 16)
```

```
    plt.tight_layout()
    plt.savefig(output_path, dpi=300, bbox_inches="tight")
    plt.show()
    plt.close()
    print(f"  saved: {output_path}")

    return reg_b, reg_a

# 5. Descriptive statistics and correlation analysis

def print_descriptive_statistics() -> None:
    """Print mean, median, quartiles, and standard deviation for each KPI."""

    print("\n=== Descriptive statistics ===")
    for label, ds in [("Baseline", baseline), ("AI-enabled", ai_enabled)]:
        print(f"\n{label} (n={len(ds['iterations'])})")
        for key in ("iterations", "ratio", "bugs"):
            arr = np.array(ds[key])
            print(f"  {key:11s}  "
                  f"mean={arr.mean():.3f}  "
                  f"median={np.median(arr):.3f}  "
                  f"std={arr.std(ddof=1):.3f}  "
                  f"Q1={np.percentile(arr, 25):.3f}  "
                  f"Q3={np.percentile(arr, 75):.3f}")

def print_correlation_analysis() -> None:
    print("\n=== Pearson correlations ===")
    for label, ds in [("Baseline", baseline), ("AI-enabled", ai_enabled)]:
        print(f"\n{label}:")
        for a, b in [("iterations", "bugs"),
                     ("iterations", "ratio"),
                     ("bugs",       "ratio")]:
            r, p = stats.pearsonr(ds[a], ds[b])
            print(f"  {a:11s} vs {b:11s}  r = {r:+.3f}  (p = {p:.3f})")

def print_welch_t_test() -> None:
    print("\n=== Welch's t-test: TC checklist ratio ===")
    t, p = stats.ttest_ind(baseline["ratio"], ai_enabled["ratio"],
                           equal_var=False)
    print(f"  t = {t:.3f}, p = {p:.3f}")

# 6. Entry point

if __name__ == "__main__":
    print("Generating figures...")
    build_figure_6()
    reg_baseline, reg_ai = build_figure_7()
    print_descriptive_statistics()
    print_correlation_analysis()
    print_welch_t_test()
```