Software Quality Journal (2026) 34:8
https://doi.org/10.1007/s11219-026-09744-9

RESEARCH

Automated test plan generation using large language models

Susmita Haldar[1,2] · **Luiz Fernando Capretz**[2]

Received: 3 March 2025 / Accepted: 3 February 2026

Abstract
Test managers create test plans as blueprints for effective software testing, but manual generation is time-consuming and requires expertise. While Large Language Models (LLMs) have been investigated for test-case generation in prior studies, their application to automated test plan creation remains limited and underexplored. This study investigates the use of LLMs to automate test plan generation in software testing, with a focus on compliance with established standards and customization for project-specific needs. It aims to explore not only how to test but also what to test. Test plans were generated for three open-source web applications using five different LLMs, namely GPT-4o, Cohere, Mistral, Llama3.1, and Google Gemini 1.5. Using the LangChain framework, each LLM was prompted to generate test plans. The generated plans were evaluated for standard conformity, practical feasibility, estimated testing effort, and readability. The test cases derived from the main functionalities were organized into suites in the test management tool *TestLink*, executed against the applications under test, and the outcomes were recorded as execution reports, which were extracted from the TestLink database for evaluation. From the evaluated open-source applications, it was observed that the quality of the generated test plans varied across different models, with real-world applications performing relatively better than the demo-based sample application in this empirical setting. While LLMs demonstrate potential in automating test planning, human validation is still necessary. LLMs can support test management by increasing automation and efficiency while requiring human oversight to ensure accuracy and reliability.

Keywords Software Testing · Test Plan Generation · Software Quality · Generative AI · Large Language Models · Artificial Intelligence

✉ Susmita Haldar
shaldar@fanshawec.ca

[1] School of Information Technology, Fanshawe College, London, Ontario, Canada

[2] Department of Electrical and Computer Engineering, Western University, London, Ontario, Canada

Published online: 21 February 2026

1 Introduction

Software systems are deeply integrated into daily life, making software testing essential for assessing quality and minimizing the risk of operational failures. However, testing goes beyond purely technical execution, and it requires careful planning, management, estimation, monitoring, and control (International Software Testing Qualifications Board, 2024b). Before test execution begins, it is crucial to determine what to test, when to test, and how to test. This requires test managers to define objectives, scope, test items, and a structured testing approach to allocate resources and budgets effectively.

A test plan serves as a comprehensive document outlining the strategy, objectives, scope, resources, methodology, key features, exclusions, schedule, risks, and other critical aspects of the testing process (Homès, 2012). Since testing teams rely heavily on the test plan to guide their activities, any inconsistencies or unrealistic elements can disrupt test execution, potentially impacting project timelines and software quality. Therefore, dedicating sufficient time to developing a well-structured and actionable test plan is essential for ensuring efficient and effective testing practices.

Organizations often don't want to create excessive testing documentation as these are time-consuming and tedious tasks, but proper planning can save time for actual test execution (Garousi et al., 2020; Bach et al., 2006). Various open-source and commercial tools assist with automating the test management process. For instance, ALM/Quality Center (formerly HP ALM) (Micro Focus, n.d.), TestRail (Gurock Software GmbH, 2024), and TestLink (TestLink Community, 2025) serve as centralized repositories for managing test cases, while tools like JMeter (Apache Software Foundation, n.d.) and the WebServer stress tool (Paessler, n.d.) are used for executing non-functional tests, such as performance, load, and stress testing. Functional test automation can be achieved using Selenium IDE for recording and executing test cases using recording and playback features, whereas Cypress (Cypress.io, 2025) and Selenium WebDriver, etc., can be used for creating robust automated testing solutions (Selenium Project, 2024). Appium (Appium Contributors, 2025; Katalon, 2025) can automate mobile application-related test cases. Additionally, JIRA facilitates defect tracking and management (Atlassian, 2025). Despite their capabilities in supporting different testing activities, these tools do not generate a comprehensive test plan, which typically needs to include a detailed scope, objective, approach, and schedule of the testing process.

With the recent advancements in LLMs, researchers and practitioners have been actively exploring their potential to automate various software testing tasks through textual representations. LLMs have demonstrated applications in automated test data generation (Karmarkar et al., 2024), automated unit test generation (Schäfer et al., 2024; Dakhel et al., 2024; Tang et al., 2024), and prioritizing crowdsourced test reports (Ling et al., 2025), among others. Recent evaluations have also examined the reliability and effectiveness of LLM outputs. For example, Li et al. (2025b) identified issues such as hallucination and variability in LLM-generated test cases across different tasks, while Zhang et al. (2025) focus on assertion generation and report encouraging improvements in accuracy and bug detection compared to existing baselines. In addition, Wang et al. (2025) introduced the TESTEVAL benchmark to systematically compare model performance on test generation tasks. These efforts, however, are largely concentrated on unit and API level test generation. In contrast, our study investigates functional, feature-oriented test planning, which empha-

sizes higher-level management aspects such as scope definition, resource allocation, and scheduling–areas that remain underexplored. Despite growing progress in LLM-driven test artifact generation, the application of LLMs in comprehensive test plan generation, which integrates both technical and managerial considerations, remains largely unexplored.

Although a test plan serves as a roadmap for guiding testing activities, creating such a document requires substantial analytical effort from experienced test leads. Test planning involves identifying testing scope, defining test items and objectives, determining appropriate testing levels and types, estimating effort, assessing risks, and deciding how testing cycles should be structured (International Software Testing Qualifications Board, 2024a). When domain knowledge or system familiarity is limited, outlining these aspects becomes particularly challenging, as test leads must reason about application behavior, workflows, and risks from minimal information. Manually performing these activities requires substantial time and domain expertise (Alagarsamy et al., 2025).

By leveraging LLMs for test plan generation, this study aims to reduce the initial manual effort associated with early-stage test planning activities, particularly the brainstorming, structuring, and documentation of core testing elements. Specifically, automation supports the rapid identification of testing scope, objectives, test items, and required test plan sections, which are traditionally defined manually by test leads. Rather than requiring these components to be developed from scratch, LLM-generated test plans provide a structured baseline that consolidates key planning information into an initial coherent document. This baseline can subsequently be reviewed, refined, and validated by the test lead. In this context, automation does not replace expert judgment or domain reasoning; instead, it reduces repetitive documentation effort and shortens the time required to produce an initial comprehensive test plan, thereby lowering the cognitive and time burden during the early phases of test planning.

Rather than relying on the front-end interface for text-based prompting, this study utilized API capabilities to enhance flexibility and enable process automation. LLMs provide robust API functionalities (OpenAI, 2024a) that can be integrated into software systems to achieve full automation. These APIs facilitate the programmatic generation of prompts and automated processing of responses, eliminating the dependency on continuous human input and advancing the automation of tasks. In this work, by integrating the API into the workflow, prompts were generated programmatically, and responses were processed seamlessly. This approach significantly reduced the need for manual intervention to generate test plans from five different LLMs, streamlining the testing process and enabling greater scalability and efficiency.

This work contributes to the research community by answering the following research questions:

RQ1: Can multiple LLMs generate relevant and complete test plans based solely on limited available information, such as the application name and basic repository details, without access to detailed application artifacts?

In this study, test plans are considered at the functional or feature level, focusing on core functionalities identified without access to full code or detailed documentation. In the early stages of software development, particularly in agile contexts, test managers often draft high-level test plans based on limited available information. This research question investigates whether LLMs, as pre-trained models with access to vast amounts of knowledge, can produce comprehensive test plans under such condi-

tions, focusing on their ability to include sprint planning and address both functional and non-functional testing requirements within the scope of testing. Additionally, given the known limitations of LLMs, such as generating hallucinated or speculative content (Waldo & Boussard, 2024; Leiser et al., 2024; Yang et al., 2024), the study examines whether the generated plans align with realistic expectations or introduce irrelevant details due to limited access to repository-specific data.

RQ2: Do LLM-generated test plans include all required components when following a standard or template, and how well do different models adhere to these structures?

This evaluation emphasizes adherence to recognized standards such as ISO/IEC/IEEE 29119 and commonly used organizational templates to determine structural completeness. This research question evaluates how effectively multiple LLMs adhere to predefined test plan templates or established standards, ensuring that all necessary components are included in the generated output. Many organizations rely on custom templates or standardized frameworks to create comprehensive test plans. This study examines whether LLM-generated test plans meet these expectations and identifies whether any specific model performs better in adhering to the required structure and completeness.

RQ3: Are test plans generated by different LLMs consistent, readable, and practically applicable when organized and executed in a test management tool such as TestLink?

Execution in this context refers to the manual testing of the system under test (SUT) based on model-generated test plans. Rather than focusing on unit-level or end-to-end testing, this process emphasizes feature-level functionality by organizing generated test cases into logical test suites within TestLink. The primary objective is to evaluate whether each model produces a coherent and easy-to-follow structure that is practical for real-world testing practice. During this phase, execution outcomes are recorded, and test cases are updated based on manual results to ensure that the test plans remain consistent, readable, and usable for effective test management.

By addressing these questions, this study contributes to the research community's understanding of the effectiveness of large language models in the context of test plan generation from the selected three different open-source applications under limited-input conditions.

The literature survey is presented in Section 2, followed by the methodology of this study in Section 3. The experimental results are presented in Section 5, followed by discussion and analysis in Section 6. Section 7 discusses the threat to the validity of this work. Finally, conclusions and future work are outlined in Section 8.

2 Literature survey

A structured test plan is essential for defining testing objectives, resources, and timelines. A well-defined test plan can lead to more efficient testing processes and better resource allocation (Chan & Tsi, 2024). Proper planning helps ensure the testing team stays focused on testing critical areas, manages resources effectively, and avoids unnecessary delays (Nguyen et al., 2003; Myers et al., 2011). Over time, test planning approaches have evolved significantly in response to changing software development methodologies.

2.1 Test plans following recognized standards

IEEE has developed several standards, including IEEE 829-1998 (IEEE Standards Committee, 1998), IEEE 829-2008 (Institute of Electrical and Electronics Engineers, 2008), and ISO/IEC/IEEE 29119 (Institute of Electrical and Electronics Engineers, 2013), outlining the evolution of the content of the test plan. The earlier standards, like IEEE 829-1998, focused on detailed, documentation-heavy test plans suitable for traditional waterfall projects. In contrast, later standards, such as ISO/IEC/IEEE 29119, emphasize flexibility for agile and risk-based testing, with ongoing planning and adjustments throughout the project. These newer standards support iterative and agile processes, allowing for adaptable test plan content.

In SWEBOK 4.0 (IEEE Computer Society, 2024), the test plan is positioned within test management as an evolving document that guides the entire testing process. It serves as a foundational element in defining the testing scope, objectives, schedules, resources, and risk management strategies while also adapting continuously to changes throughout the project lifecycle. This dynamic approach aligns with agile and iterative practices, treating the test plan as a living document that integrates test monitoring and control activities. The test plan not only sets the initial testing direction but also supports ongoing adjustments based on testing outcomes, ensuring that test management remains flexible and responsive to project needs.

2.2 Test plan generation techniques and challenges

As web applications continue to evolve, Balsam and Mishra (2025) examined key challenges in web application testing. They highlighted obstacles such as cross-browser compatibility, dynamic content handling, and security vulnerabilities, all of which complicate the testing process. Their work underscores the need for robust test-planning strategies that ensure software reliability.

In the context of test planning, Heusser and Larsen (2023) emphasize the importance of integrating test planning into the overall software development process. They advocate for a risk-based approach, where teams prioritize testing efforts by identifying potential risks early. They also highlight the importance of flexible test plans that adapt to evolving project requirements and emphasize collaboration among team members to ensure that diverse perspectives are considered.

Structured planning approaches have been used to generate test cases for both functional and non-functional testing by defining pre- and post-conditions within a structured model by Bozic and Wotawa (2019). These methods enable flexibility by modifying conditions dynamically, making them effective for functional testing. However, they require predefined rules and struggle with handling parameter values, often necessitating additional effort to refine test cases.

We assess how well LLM-generated test plans align with structured templates, covering functional and non-functional aspects of testing, including scope, execution time estimates, and overall relevance for real-world test planning.

Alternative structured approaches, such as Lean Canvas-based models, have been explored to improve test planning, particularly for mobile applications. Nidagundi and Novickis (2017) proposed a Lean Canvas framework to enhance test strategy by organiz-

ing functional and non-functional testing into a structured visual board. Their method helps streamline test planning, risk identification, and collaboration, ensuring comprehensive test coverage. While Lean Canvas provides a structured way to improve test strategy, it relies on predefined visual templates. Our study, instead, evaluates whether LLMs can generate structured test plans dynamically using minimal input. By comparing these approaches, we assess the flexibility and completeness of LLM-generated test plans in addressing testing requirements.

Forgács and Kovács (2024) introduced model-based testing approaches to enhance efficiency and effectiveness in test planning. Their work highlights the role of risk analysis in prioritizing testing efforts and emphasizes advanced techniques that refine test plans, making them more relevant in complex software environments. They also stress the importance of collaboration between developers and testers in refining test plans.

While these studies provide valuable insights into manual test planning approaches, recent advancements in LLMs suggest the potential for automating parts of the test planning process.

Industry has also begun formalizing guidance in this area. The ISTQB syllabus on Generative AI in Software Testing (International Software Testing Qualifications Board, 2025) explicitly identifies test planning as a potential application of LLMs but notes the lack of empirical validation. Winteringham (Winteringham, 2024) also highlights the emerging role of generative AI in supporting structured testing practices.

2.3 Instruction-based prompt engineering technique

Prompt engineering is a technique for designing and optimizing inputs to enhance the effectiveness of LLMs in generating precise responses. Well-structured prompts improve an LLM's ability to handle a wide range of tasks, from question answering to arithmetic reasoning (Prompt Engineering Guide, 2025). A standard prompt usually has the format of a question or instructions.

Prompting techniques can be categorized into zero-shot prompting, which provides instructions without example (Kojima et al., 2022), few-shot prompting, which supplies a few examples to guide the model (Ma et al., 2023), and chain-of-thought prompting, which encourages step-by-step reasoning (Wei et al., 2022). In this study, we employed instruction-based prompting with a combination of zero-shot and few-shot approaches. While some prompts included only the standard name following the zero-shot prompting technique, others referenced specific sections from the standard corresponding to the few-shot prompt technique to enhance precision.

Ouyang et al. (2022) demonstrated that instruction-based fine-tuning with human feedback is a promising approach for aligning language models with human intent. Similarly, Pornprasit and Tantithamthavorn (2024) analyzed LLM-based code review automation in two contexts: fine-tuning (training the model on a domain-specific dataset) and prompting (guiding the model's response through structured instructions without retraining). Their findings highlight that well-crafted prompts can significantly enhance an LLM's performance without requiring additional model training.

This aligns with Schulhoff et al. (2024), who provided a systematic survey of prompting techniques and showed how different strategies affect LLM performance. Since prompt

design strongly influences test artifact quality, our study adopts instruction-based prompts aligned with testing standards.

2.4 LLMs in software testing

LLMs have been applied in other test artifact generation tasks, such as unit test generation (Schäfer et al., 2024) and user acceptance test case generation. Zhang et al. (2025) explored assertion generation with LLMs, while Alagarsamy et al. (2025) enhanced text to test case generation through prompting strategies. Saboor Yaraghi et al. (2025) and Tip et al. (2025) extended this research into automated test case repair and mutation testing. These works demonstrate the breadth of LLM applications in test artifact generation.

Belzner et al. (2024) explored the challenges of using LLMs in software engineering tasks. Their study compared Bard and ChatGPT for unit and system test generation, where Bard produced more abstract test cases while ChatGPT generated more concrete ones. Similarly, Karpurapu et al. (2024) demonstrated that GPT-3.5 and GPT-4 produced high-quality, error-free BDD acceptance tests, with few-shot prompting improving accuracy.

Beyond unit- and system-level studies, emerging research has begun exploring feature- and functionality-level test generation using LLMs. Ferreira et al. (2025) conducted an industrial case study demonstrating how LLMs can derive acceptance-level Gherkin scenarios from user stories, while Milchevski et al. (2025) proposed a multi-step framework for system-level test specification generation from textual requirements. Similarly, Li et al. (2025a) investigated functional web-form test generation across multiple open-source applications, highlighting LLMs' growing ability to capture higher-level interactions and intent. These studies collectively indicate a shift toward end-to-end and feature-oriented test artifact generation, whereas the present work focuses on LLM-based test plan generation, encompassing objectives, scope, risks, and feature coverage as defined in ISO/IEC/IEEE 29119-3.

Comprehensive evaluations such as Li et al. (2025b) and Wang et al. (2025) emphasize benchmarking across models, confirming both opportunities and limitations of LLM-driven test generation. Wang et al. (2025) further situate LLMs within broader agent-based software engineering workflows, suggesting future directions for integrating planning and reasoning. While these contributions enrich our understanding of LLM performance, they primarily address unit or system-level test generation, leaving the domain of structured test plan creation comparatively unexplored.

This gap is also acknowledged in the practitioner literature. Winteringham (2024) highlights the emerging role of generative AI in structured testing practices, while the ISTQB CT-GenAI syllabus (International Software Testing Qualifications Board, 2025) identifies test planning as a potential LLM application but stresses the lack of empirical validation. Together with recent surveys such as Wang et al. (2024), these sources emphasize that while LLMs are increasingly validated for test generation, their use in higher-level planning remains underexplored.

Despite their success in test case automation, LLMs have not been widely used for test planning. Santos et al. (2024) surveyed industry professionals and found that 40% of the respondents used LLMs during the initial phases of software testing, including requirements analysis, test design, and test plan creation. Wang et al. (2024) identified that LLMs have not

been widely adopted for test planning, primarily due to the need for human expertise and the lack of publicly available data for training.

Unlike test case generation, which follows structured templates, test planning requires strategic decision-making, risk assessment, and project alignment. These characteristics increase the complexity of test planning and highlight the need for a systematic evaluation of LLM-generated test plans.

While LLMs offer promising advancements in test planning automation, concerns regarding bias, transparency, and accountability remain underexplored. Strandberg et al. (2011) discussed ethical concerns in AI-driven regression test selection (RTS), identifying challenges such as assigning responsibility, bias in decision-making, and lack of participation in AI-generated decisions. They propose solutions, including explicability, human supervision, and diversity in AI-driven testing models.

These ethical concerns are also relevant in the context of LLM-generated test plans, where AI-generated outputs may suffer from hallucinated test objectives, missing key test components, or biases in selection criteria. This highlights the need for systematic evaluation to ensure that LLM-generated test plans are accurate, reliable, and ethically sound. These concerns are also consistent with Gallegos et al. (2024), who surveyed bias and fairness in LLMs, and Zhao et al. (2026), who reviewed retrieval augmented generation as a strategy to improve reliability.

2.5 Readability of the generated test plans

The readability and understandability of test code are crucial aspects of software engineering research. Winkler et al. (2024) investigated test code readability, while Setiani et al. (2020) developed a test case understandability model to assess comprehension.

Readability metrics have been widely applied to measure textual complexity. The Gunning Fog Index (Gunning, 1952) estimates the years of formal education required to understand a text. This score directly reflects the approximate grade level required for reading comprehension. A higher fog index indicates a more complex text, with a score of 12 corresponding to the reading level of a high school senior. In an educational system, students up to Grade 8 are generally considered to be at an elementary or middle school level. Grades 9 to 12 correspond to high school education, while Grades 13 to 16 represent undergraduate-level study. Values above Grade 16 indicate graduate-level reading complexity. Based on this interpretation, Fog Index scores were categorized for this study as very easy up to Grade 8, standard for Grades 9 to 12, difficult for Grades 13 to 16, and very difficult above Grade 16.

Similarly, the Flesch-Kincaid Grade Level (Flesch, 1948) calculates readability based on sentence length and word complexity. A score of 9.3 suggests that a ninth grader can comprehend the text. The Flesch Reading Ease Score (FRES) (Kincaid et al., 1975) provides another measure, where higher scores indicate easier readability.

Readability metrics have been used in various domains. Zhou et al. (2017) evaluated the consistency of readability equations for assessing design standards. Beyoğlu et al. (2024) applied readability metrics to assess the quality of ChatGPT-generated medical information, while Cocci et al. (2024) examined the appropriateness of ChatGPT-generated content for medical patients. In software engineering, Yeow et al. (2024) utilized Flesch readability metrics to validate software requirements engineering using GPT-3.5.

Our study will consider this readability aspect as one of the evaluation criteria of the generated test plans.

2.6 Research contribution

This study contributes to the research community by evaluating the feasibility of LLM-driven test plan generation, specifically:

- Assessing whether multiple LLMs can generate relevant and complete test plans.
- Examining how effectively different LLMs adhere to established test plan templates and structures, ensuring completeness and compliance with predefined formats.
- Evaluating the consistency, readability, and practical applicability of LLM-generated test plans when used in test management tools such as TestLink.
- Comparing multiple LLMs across three open-source applications to assess their effectiveness in generating structured and actionable test plans.

By addressing these aspects, this study provides insights into the capabilities and limitations of LLMs in test planning, identifying key areas where human intervention remains necessary and where AI-driven automation can enhance the test management process. Unlike studies focused solely on unit or module level test case generation, the present work targets feature level planning, where each generated test plan encompasses multiple testing levels, including functional, integration, system, and acceptance testing as part of ISO/IEC/IEEE 29119-3 compliance.

3 Methodology

This study explores the application of LLMs in automated test plan generation for open source software applications. We utilized four freely accessible LLMs: Cohere, Mistral, Llama3, and Google Gemini, along with OpenAI GPT-4o, which required commercial API access. These models generated structured test plans for three diverse open source applications: OrangeHRM (HR management system), ZenCart (e-commerce platform), and Spring PetClinic (pet clinic management demo application). LLMs are invoked via APIs using standardized test plan templates, with the user able to specify a preferred standard such as IEEE 829 Institute of Electrical and Electronics Engineers (2008) or IEEE/ISO/IEC 29119-3:2021 International Organization for Standardization, International Electrotechnical Commission, IEEE (2021). For the evaluations in this work, IEEE/ISO/IEC 29119-3:2021 was selected as the reference standard for validation.

The overall methodology follows a structured process where user inputs are collected via a web interface, processed by multiple LLMs through LangChain, and stored systematically in text and CSV formats for further analysis and execution. The extracted test cases are then exported to TestLink for validation. This workflow ensures structured, repeatable, and analyzable test plan generation, as illustrated in Fig. 1.

This figure presents the overall architecture of the LLM-based test plan generation framework. The workflow begins by providing two key inputs to the system: the GitHub repository URL of the selected open-source project and the desired test plan type, aligned

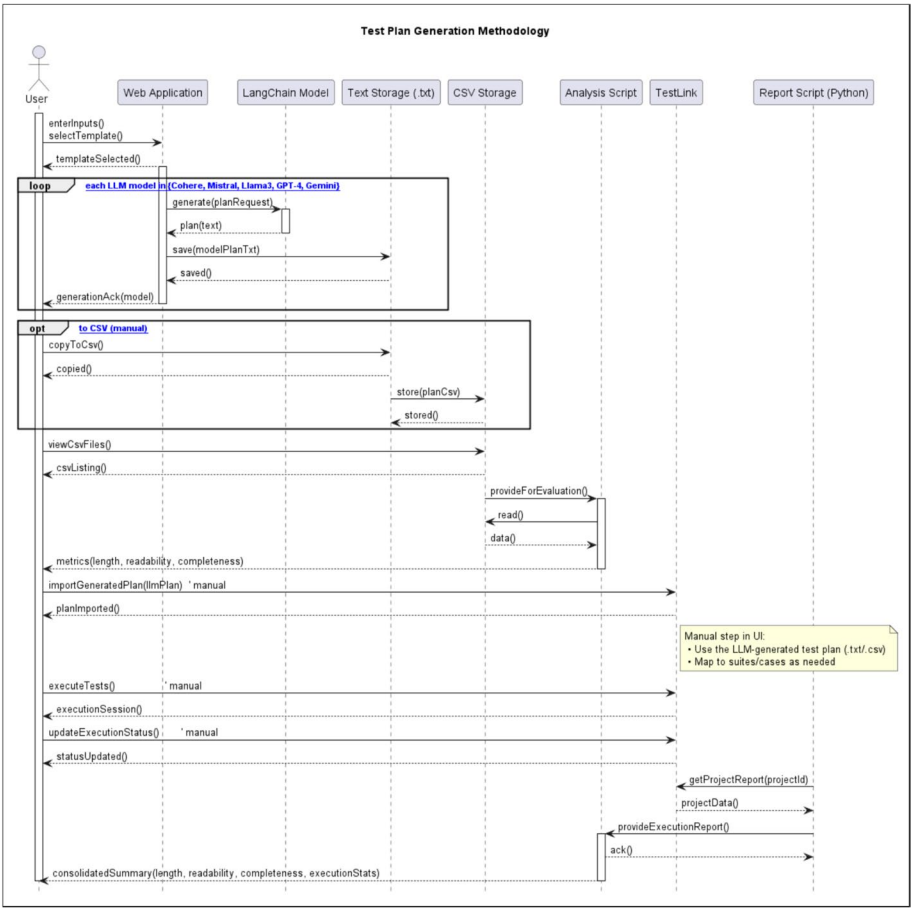


Fig. 1 Methodology of generating test plans using large language models

with ISO/IEC/IEEE 29119-3:2021. These inputs are passed through a LangChain-based API layer that interfaces with multiple LLMs consisting of Cohere, Mistral, GPT-4o, Gemini, and Llama3. The models do not read or parse the source code. Instead, the repository link serves purely as a contextual reference within the structured prompts. Figure 1 illustrates how the prompt generation component dispatches requests to each LLM, while Algorithm 1 and Fig. 3 further describe the automated construction, execution, and storage of test plans through this API pipeline.

The user interface of the proposed system includes an option to provide detailed application descriptions as input. However, for this study, we intentionally used only the GitHub repository link to evaluate how effectively each LLM could generate structured and accurate test plans from publicly available information alone. This setup reflects a limited input scope designed to assess model capability under realistic data constraints. If the generated test plans are comprehensible and ISO/IEC/IEEE 29119-3 compliant in this minimal-input setting, it can be inferred that providing richer contextual information, such as requirements,

architecture details, or user stories, would likely result in even more complete and robust outputs.

3.1 Data collection and model setup

The selected applications span diverse software domains, architectures, and programming paradigms, ensuring a robust evaluation of LLM-generated test plans in different real-world contexts. Table 1 summarizes their key attributes.

Each column in the table provides essential repository insights. The Project Name identifies the software under test, while Open Issues indicates the number of unresolved issues, reflecting ongoing development activity. The Watchers metric represents the number of GitHub users monitoring the repository, whereas Forks denotes instances where developers have cloned the repository to create modified versions. The Stars metric serves as an indicator of project popularity, representing how many users have liked the repository. The 'Primary Language' column lists the dominant programming language used, while the 'Other Languages' column provides a percentage-based distribution of additional programming languages, highlighting the project's complexity. Lastly, the Type column describes the application's primary purpose.

The selected applications ensure diverse software representation, allowing for a broad evaluation of LLM-generated test plans. Spring PetClinic (Spring Team, 2025) is a widely used reference project for Java Spring applications, serving as a benchmark for test automation and model-based testing (Yeh et al., 2024; Leotta et al., 2024). Given its structured implementation and well-documented features, it provides an ideal environment for assessing LLM-driven test plan generation in Java-based applications.

ZenCart (ZenCart Developers, 2025), a real-world PHP-driven e-commerce platform, has been utilized in research on testing challenges in web applications (Xu et al., 2016; Grigera et al., 2016). Its dynamic nature and modular architecture make it a suitable candidate for evaluating LLM-generated test plans for complex e-commerce workflows.

OrangeHRM (OrangeHRM Team, 2025), a comprehensive enterprise HR management system, introduces additional complexities in business application testing. Researchers have applied this application in model-based testing (Schur et al., 2015), configuration testing (Patel & Shah, 2015), and crawlability assessments (Marchetto et al., 2011), demonstrating its suitability for evaluating test plan generalization in enterprise environments.

Table 1 Overview of software under test used in this study

Project Name	Open Issues	Watchers	Forks	Stars	Primary Language	Other Languages	Type
Spring PetClinic	8	383	24,387	7,945	CSS	Java, HTML, SCSS, Dockerfile	Pet Clinic Application
ZenCart	80	55	236	384	PHP	JavaScript, CSS, HTML, Hack	E-Commerce Platform
OrangeHRM	94	47	585	878	PHP	Vue, TypeScript, SCSS, JavaScript	HR Management System

Notes. Data collected from the public GitHub repositories of the selected projects as of January 2025

By selecting applications from distinct software domains, programming paradigms, and established research contexts, this study ensures a comprehensive assessment of LLM-driven test plan generation, focusing on its generalizability and robustness across varying system complexities and testing requirements.

Figure 2 provides an overview of the user interface dashboards for each software application, illustrating their structure and functionality. These visuals help contextualize the testing environments where LLM-generated test plans are applied.

3.2 LLMs used for generating test plans

This study explores the application of LLMs for automated test plan generation using repositories from GitHub. Five different models, consisting of Mistral, GPT-4, Cohere, Llama3, and Gemini, were integrated using LangChain, a framework that enables seamless interaction with multiple LLMs through a standardized API interface. The goal is to evaluate the effectiveness, consistency, and adaptability of these models in generating structured and comprehensive test plans.

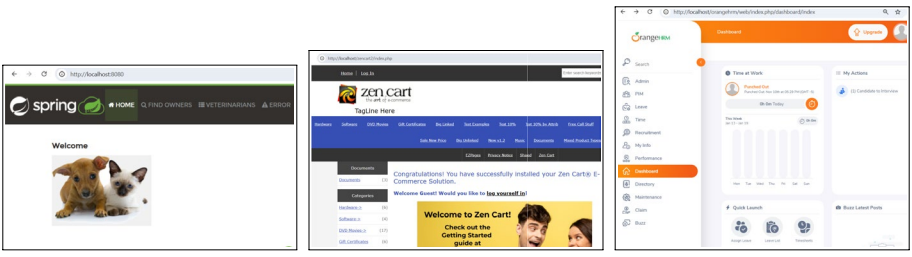
The modular implementation allows models to be dynamically invoked based on availability and suitability. LangChain simplifies this process by offering prebuilt integrations, structured outputs, and automation capabilities, enabling efficient test plan generation at scale. Table 2 provides an overview of the specific LangChain integrations used for each model.

3.3 LLMs utilized in this study

The following LLMs were utilized in this study. Mistral (Mistral, 2024), GPT-4o (OpenAI, 2024b), Cohere (Cohere, 2025). llama-3.1-8b-instant variant via Groq API (Groq, Inc., 2025), Google Gemini (Pichai & Hassabis, 2024).

3.4 LangChain integration with LLMs

Each selected LLM has unique capabilities that contribute to the test plan generation process. By utilizing LangChain's chat model framework, this study optimizes response processing efficiency while maintaining uniformity in test plan generation.



(a) Main Menu of Spring-Clinic Application (b) Main Menu of ZenCart Application (c) Main Menu of OrangeHRM Application

Fig. 2 Software under test for validating the generated test plans

Table 2 LangChain integration with LLM models

LLM Model	LangChain Integration (Import)	API Used	Model Version
Cohere	`from langchain_cohere.chat_models import ChatCohere`	`Cohere API`	`command`
Mistral	`from langchain_mistralai import ChatMistralAI`	`Mistral API`	`mistral-large-latest`
Llama3	`from langchain_groq import ChatGroq`	`Groq API`	`llama-3.1-8b-instant`
GPT-4	`from langchain_openai import ChatOpenAI`	`OpenAI API`	`gpt-4o`
Google Gemini	`from langchain_google_genai .chat_models import ChatGoogleGenerativeAI`	`Google GenAI API`	`gemini-1.5-flash-001`

3.5 TestPlan generation

To ensure comprehensive test plan generation, our system integrates multiple LLMs, each producing a unique test plan based on a standardized template. The system allows selecting from IEEE 829, ISO/IEC/IEEE 29119-3:2021, and other applicable templates. However, this study focuses on validating test plan generation conforming to the ISO/IEC/IEEE 29119-3:2021 template to maintain a well-defined scope and enable concise and effective validation. Validation of other templates would follow the same approach. To enhance usability, the system is deployed as a web-based application using Streamlit. Users can select the desired model, specify project details, and generate test plans dynamically. The interface allows for seamless interaction and retrieval of test plans. A screenshot of the interface is shown in Fig. 3.

The workflow involves retrieving the necessary inputs, invoking LLMs, and storing the generated test plans for evaluation. The required inputs include the application name, the source type or link (such as a GitHub repository or project documentation), the chosen test plan template, and the project start date. Two additional optional inputs are the team size and any reference content. Team size refers to the number of people involved in the testing effort, which may influence workload distribution and scheduling. Reference content refers to supplementary material that provides additional context for generating the test plan, such as requirement documents, design specifications, or user stories.

For generating test plans, the LLMs were provided only with the application name and GitHub repository link as contextual input. The repository link was treated purely as a string in the prompt, and no source code, metadata such as stars, forks, issues, or watchers, or supplementary files were included. This ensured that the generation relied solely on application-level context rather than detailed repository artifacts. The overall process is described in Algorithm 1. Although the primary focus of this study is feature-level test planning, the generated artifacts inherently address multiple testing levels, including functional, integration, system, and acceptance testing within each plan's scope section.

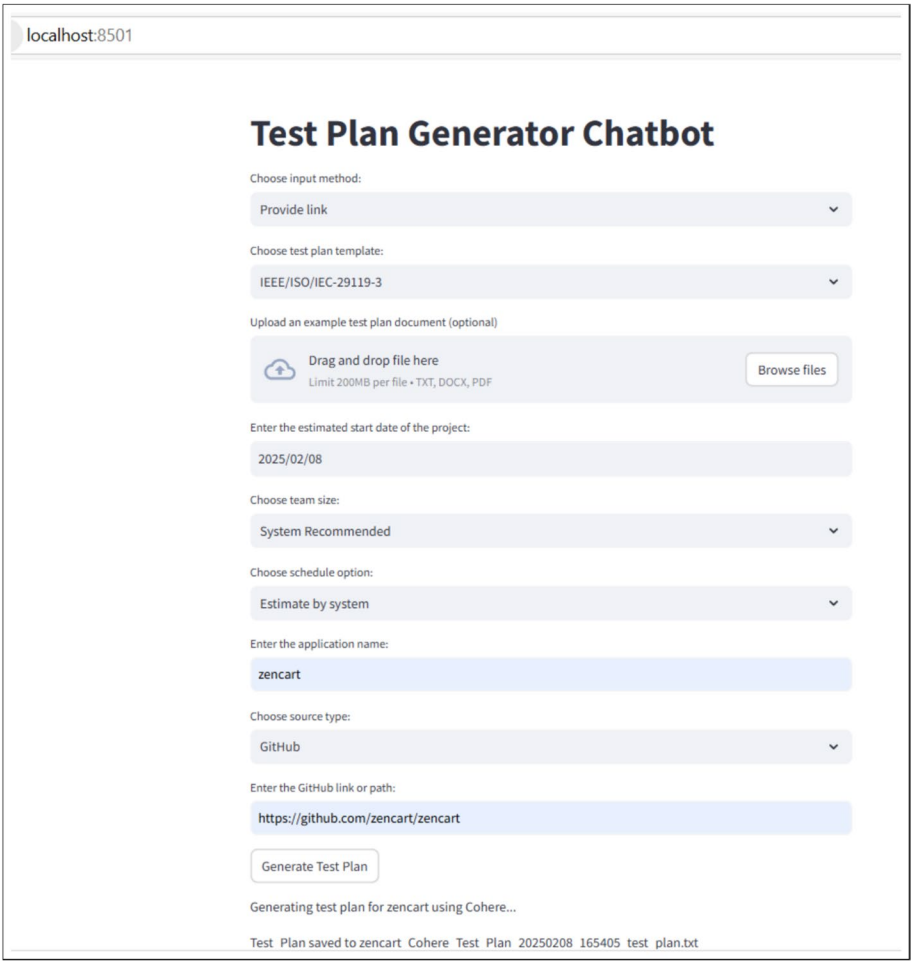


Fig. 3 User interface of the multi-model test plan generator

Algorithm 1 Multi-model test plan generation workflow

1: **Input:** Application name, source type or link, selected template, start date, team size (optional), reference content (optional)
2: **Output:** Generated test plans per model
3: Retrieve template prompt using `template_choice`
4: Call **Multi-Model LLM-Based Test Plan Generation**
5: Print confirmation message

The function responsible for generating test plans invokes multiple LLMs based on user selection. The LLM is initialized with its respective API key, and a formatted prompt is constructed dynamically. The selected model processes the input and generates a structured test plan. The process includes optional handling of team size and reference content if pro-

vided by the user, allowing for more contextually relevant output. The detailed process is described in Algorithm 2.

Algorithm 2 Multi-model LLM-based test plan generation

Step 1: Load Environment and API Key
Step 2: Select LLM Model
if `llm_model` = `"GPT-4o"` **then**
    Use `ChatOpenAI`.
else if `"Cohere"` **then**
    Use `ChatCohere`.
else if `"Mistral"` **then**
    Use `ChatMistralAI`.
else if `"Llama3"` **then**
    Use `ChatGroq`.
else if `"Gemini"` **then**
    Use `ChatGoogleGenerativeAI`.
else
    Return "Error: Unsupported model".
end if
Step 3: Set Parameters – `temperature` = 1.0, `max_tokens` = 4000.
Step 4: Validate Inputs and Format Prompt
if `prompt` or `github_link` is empty **then**
    Return "Error: Missing input".
else
    Replace {`github_link`} in `prompt`.
end if
: Include team size and reference content in prompt if provided.
Step 5: Invoke LLM and Process Response
Send prompt to model and store `response`.
if `response` has `content` **then**
    Return `response.content`.
else if `response` is a list **then**
    Concatenate and return.
else
    Return "Error: Unexpected format".
end if
Step 6: Handle Errors
if exception occurs **then**
    Return error message.
end if

The test plans were generated using the ISO/IEC/IEEE 29119-3:2021 template with three different prompting approaches. The first approach used the template as-is, while the second specified the important sections along with some additional sections. The third approach provided explicit specifications for all sections. These variations of these prompts are described in Section 4.

4 Prompt templates

This study evaluates three distinct prompt styles for generating ISO/IEC/IEEE 29119–3:2021–compliant test plans, each differing in the level of structural guidance provided to the LLM. These prompt templates were designed to represent a progression from minimal

guidance (Prompt 1) to moderate contextualization (Prompt 2) and finally to complete structural enforcement (Prompt 3), enabling a controlled comparison of LLM behavior under varying levels of prompt constraint. The detailed content of each prompt is provided in the following subsections, and each prompt is later assessed for section-level test plan coverage, as discussed in Section 5.1.

4.1 Prompt 1 - Functional test plan prompt

The Functional Test Plan Prompt requests the generation of a complete test plan using only the template name and project-related information, without explicitly listing section headings, while instructing the model to include all sections defined in the standard (Fig. 4).

Functional Test Plan Prompt

Create a comprehensive test plan based on the International Standard ISO/IEC/IEEE 29119-3:2021 — *Software and Systems Engineering — Software Testing — Part 3: Test Documentation.*
This test plan must strictly follow the prescribed structure from the standard and include all required sections without modifications, omissions, or merging of content. **Strict Compliance Required:** Do not rename, omit, or merge any sections. Each section must be explicitly included as listed above.

Fig. 4 Prompt for generating functional test cases based on ISO/IEC/IEEE 29119-3

4.2 Prompt 2 - Comprehensive test plan prompt

The Comprehensive Test Plan Prompt includes the template name along with a limited set of additional contextual sections beyond those specified in the standard, enabling controlled customization while still requiring the inclusion of all mandatory ISO/IEC/IEEE 29119–3 sections (Fig. 5).

Comprehensive Test Plan Prompt

Create a comprehensive test plan based on the IEEE/ISO/IEC 29119-3:2021 standard. Include sections such as Introduction, Scope, References, Glossary, Test Item(s), Test Scope, Assumptions and Constraints, Stakeholders, Testing Communication, Risk Register, Product Risks, Project Risks, Test Strategy, Test Deliverables, Test Design Techniques, Test Completion Criteria, Metrics to be Collected, Retesting, Suspension and Resumption Criteria, Deviations from Organizational Test Practices, Testing Activities and Estimates, Staffing, Hiring Needs, Training Needs, and Schedule. Also include a list of main functionalities and example test cases for each functionality. Additionally, add a section called **Additional Information** detailing the recommended number of sprints for agile testing and the tasks for each sprint.

Fig. 5 Template prompt for generating a full IEEE/ISO/IEC 29119-3:2021 compliant test plan

4.3 Prompt 3 - The section-enforced test plan prompt

The Section-Enforced Test Plan Prompt explicitly lists all standard section headings, ensuring that the generated test plan strictly follows the prescribed structure (Fig. 6). To support reproducibility, a replication package has been made publicly available via Figshare at https://figshare.com/s/243347265e51fabed262. The package contains the full implement ation used in the experimental evaluation, including model-specific wrappers for invoking five large language models (GPT-4, Cohere, Mistral, Llama 3, and Gemini) and three main applications corresponding to the evaluated prompt strategies. Each main application internally constructs the natural-language prompt based on the selected prompt type (minimal guidance, partial structural guidance, or full ISO/IEC/IEEE 29119-3 section enforcement). The GitHub repository link of the target system is passed as a runtime variable and programmatically inserted into the prompt before model invocation.

In this study, we intentionally restricted the input provided to the LLMs to a GitHub repository link. This design choice was made to evaluate the feasibility of generating high-level test plans under minimal-information conditions, which commonly occur during early phases of software testing when detailed requirements or formal documentation are not yet available. While the proposed framework supports richer inputs such as requirements specifications or application descriptions, these were excluded to isolate the impact of prompt structure and standard enforcement.

The Section-Enforced Test Plan Prompt

Create a comprehensive test plan based on the International Standard ISO/IEC/IEEE 29119-3:2021 — *Software and Systems Engineering — Software Testing — Part 3: Test Documentation*. This test plan must strictly follow the prescribed structure from the standard and include all required sections without modifications, omissions, or merging of content.

Required Sections (ISO/IEC/IEEE 29119-3:2021):

7.2.2 Context of testing
7.2.2.1 Projects / test levels / test types
7.2.2.2 Test items
7.2.2.3 Test scope
7.2.2.4 Test basis
7.2.3 Assumptions and constraints
7.2.4 Stakeholders
7.2.5 Testing communication
7.2.6 Risk register
7.2.6.1 General
7.2.6.2 Product risks
7.2.6.3 Project risks
7.2.7 Test strategy
7.2.7.1 General
7.2.7.2 Test levels
7.2.7.3 Test types
7.2.7.4 Test deliverables
7.2.7.5 Test design techniques
7.2.7.6 Entry and exit criteria
7.2.7.7 Test completion criteria
7.2.7.8 Degree of independence
7.2.7.9 Metrics to be collected
7.2.7.10 Test data requirements
7.2.7.11 Test environment requirements
7.2.7.12 Retesting
7.2.7.13 Regression testing
7.2.7.14 Suspension and resumption criteria
7.2.7.15 Deviations from the organizational test practices
7.2.8 Testing activities and estimates
7.2.9 Staffing
7.2.9.1 General
7.2.9.2 Roles and responsibilities
7.2.9.3 Hiring needs
7.2.9.4 Training needs
7.2.10 Schedule

Strict Compliance Required: Do not rename, omit, or merge any sections. Each section must be explicitly included as listed above.

Fig. 6 Prompt for generating test plan following ISO/IEC/IEEE 29119-3:2021 by passing all the required sections

This experimental setup also enables observation of how pretrained knowledge embedded within different LLMs influences test plan generation when external contextual input is intentionally minimized. By limiting supplementary artifacts, variations in output quality may reflect differences in model pretraining and internal representations. Although this study does not aim to quantify the effects of pre-trained knowledge, it enables a qualitative comparison of LLM behavior under identical input constraints. The replication package also includes execution instructions and configuration templates for API keys. As large language model providers periodically update or deprecate model identifiers, users reproducing the experiments may substitute equivalent available model versions while preserving the original prompt structure and experimental workflow.

4.4 Validation strategies

The validation process was designed to ensure that the generated test plans were not only structurally complete but also practical and usable in real testing contexts. To achieve this, the following aspects were evaluated:

- Ensure all the artifacts are available in all the generated test plans.
- Evaluate the readability of the generated test plans.
- Verify the testing schedule.
- Use TestLink to keep track of the test case execution results.

4.5 Readability as one of the evaluation criteria

To assess the readability of AI-generated test plans, three readability metrics will be used: Gunning Fog Index (Gunning, 1952) estimates the education level required for comprehension. The Flesch-Kincaid Grade Level (Flesch, 1948) determines readability in terms of the U.S. school grade level. The Flesch Reading Ease Score (FRES) (Kincaid et al., 1975) assigns a numerical value, with higher scores indicating greater readability.

The readability scores will be calculated using the Textstat Python library https://github.com/textstat/textstat, which automates the analysis. This ensures that AI-generated test plans are not only syntactically correct but also comprehensible to software testers.

By incorporating these readability metrics, this study aligns with prior research, such as Winkler et al. (2024) and Setiani et al. (2020), which emphasize the importance of readability and understandability in software testing. Readability scores will serve as an objective evaluation criterion, ensuring that AI-generated test plans are accessible to testers with varying levels of expertise.

4.6 Validation of test execution using TestLink report

To validate the test execution results, the testing was organized in TestLink, which systematically stores all test-related data in a structured MySQL database schema. This database structure was leveraged to extract relevant information about the test cases executed and their results. The extraction process focused on retrieving data from key functionalities to

ensure comprehensive validation. Further details regarding the extracted data and its analysis are provided in the results section, while the specific algorithm used for data extraction is described in Algorithm 3. The structured output allows subsequent analysis of execution patterns, success rates, and effort allocation.

Algorithm 3 Test execution data collection from TestLink

Step 1: Connect to the TestLink database.
Establish a connection using MySQL with authentication credentials.
Step 2: Define SQL query for data extraction.
Retrieve data from the `exec_by_date_time` table, including:

- `id` (Execution ID)
- `testplan_name` (Test Plan Name)
- `tcversion_id` (Test Case Version ID)
- `execution_ts` (Execution Timestamp)
- `execution_duration` (Execution Duration)
- `status` (mapped as: 'p' → Passed, 'f' → Failed, 'b' → Blocked, others → Not Executed)

Also fetch test case details from `nodes_hierarchy`:

- `name` where `node_type_id = 3` (Test Case Name)
- `name` where `node_type_id = 2` (Test Suite Name)

Step 3: Execute SQL and fetch results.
Step 4: Convert data to structured format using Pandas DataFrame.
Define columns: Execution ID, Test Plan, Test Case Version ID, Timestamp, Status, Duration, Notes, Case Name, Suite Name.
Step 5: Export data to CSV.
Save as `testlink_exec_report.csv`.
Step 6: Close database connection.

5 Results

This section evaluates the generated test plans based on different prompts, test execution outcomes, estimated effort, and readability analysis. The results highlight the effectiveness of different LLM models and prompt strategies in generating comprehensive test plans.

5.1 Test plan coverage across prompts

Table 3 summarizes the extent to which test plans generated by different prompts align with ISO/IEC/IEEE 29119-3:2021 test plan sections.

The results indicate that a more detailed prompt significantly improves test plan coverage. The baseline prompt, which instructed the LLM to generate a test plan following the ISO/IEC/IEEE 29119-3:2021 standard without explicitly specifying section names, resulted in 38% coverage of the standard's sections. The second prompt, which explicitly highlighted key sections to include along with some additional sections, increased coverage to 77%. Finally, the third prompt, which specified each section from the standard without modification, achieved 100% coverage, ensuring completeness across all test plan sections.

These findings suggest that providing structured and detailed input enables LLMs to generate more comprehensive test plans, reducing gaps in key areas such as risk registers, test strategy, and communication plans.

Table 3 Test plan section coverage across different prompts

ISO/IEC/IEEE 29119 Section	Prompt 1 (Minimal Prompt)	Prompt 2 (Key Fields)	Prompt 3 (Full List)
7.2.2 Context of Testing	✗	✓	✓
7.2.2.1 Projects/Test Levels/Test Types	✗	✓	✓
7.2.2.2 Test Items	✓	✓	✓
7.2.2.3 Test Scope	✓	✗	✓
7.2.2.4 Test Basis	✗	✗	✓
7.2.3 Assumptions and Constraints	✓	✓	✓
7.2.4 Stakeholders	✗	✓	✓
7.2.5 Testing Communication	✗	✓	✓
7.2.6 Risk Register	✓	✓	✓
7.2.6.1 General	✗	✗	✓
7.2.6.2 Product Risks	✗	✗	✓
7.2.6.3 Project Risks	✗	✗	✓
7.2.7 Test Strategy	✓	✓	✓
7.2.7.1 General	✓	✓	✓
7.2.7.2 Test Levels	✓	✓	✓
7.2.7.3 Test Types	✓	✓	✓
7.2.7.4 Test Deliverables	✓	✓	✓
7.2.7.5 Test Design Techniques	✓	✓	✓
7.2.7.6 Entry and Exit Criteria	✓	✓	✓
7.2.7.7 Test Completion Criteria	✓	✓	✓
7.2.7.8 Degree of Independence	✗	✗	✓
7.2.7.9 Metrics to be Collected	✗	✓	✓
7.2.7.10 Test Data Requirements	✓	✓	✓
7.2.7.11 Test Environment Requirements	✓	✓	✓
7.2.7.12 Retesting	✓	✓	✓
7.2.7.13 Regression Testing	✓	✓	✓
7.2.7.14 Suspension and Resumption Criteria	✗	✓	✓
7.2.7.15 Deviations from Organizational Test Practices	✗	✓	✓
7.2.8 Testing Activities and Estimates	✓	✓	✓
7.2.9 Staffing	✗	✓	✓
7.2.9.1 General	✗	✓	✓
7.2.9.2 Roles and Responsibilities	✗	✓	✓
7.2.9.3 Hiring Needs	✗	✓	✓
7.2.9.4 Training Needs	✗	✓	✓
7.2.10 Schedule	✓	✓	✓
Total Sections Covered	**15/39 or 38% coverage**	**30/39 or 77% coverage**	**39/39 or 100% coverage**

Notes. ✓ = section covered; ✗ = section missing

The improvement in test plan coverage with increased prompt detail was consistently observed across all evaluated LLMs, particularly for Prompt 3, which explicitly listed all ISO/IEC/IEEE 29119-3 sections. However, greater variability was observed across models

for Prompt 1 and Prompt 2, where the level of guidance was lower, and interpretation was more open. This variability, as reflected in Table 3, may be influenced by differences in model capacity and prior knowledge, which can affect how incomplete or partially specified prompts are interpreted. As prompt specificity increases, these model-dependent differences are reduced, resulting in more uniform coverage outcomes.

5.2 Test execution results

Although Prompt 3 included all ISO/IEC/IEEE 29119-3 sections, including Test Items, it did not ensure that the core functionalities of the target application were explicitly identified and linked to corresponding test cases. In contrast, Prompt 2 specifically required the listing of main functionalities to be tested along with example test cases for each functionality. This application-specific detail was essential for validating not only structural compliance but also the practical relevance of the generated test plans. Therefore, Prompt 2 was selected for test execution validation.

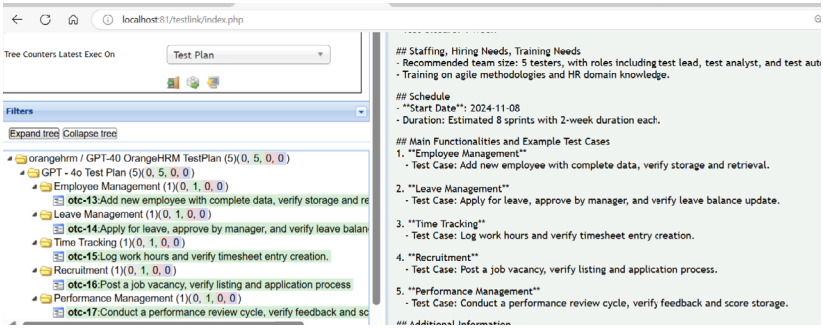
To streamline the organization of testing activities and facilitate efficient test execution result collection, the test management tool TestLink was used. A separate test project was created for each selected application, with each test project containing five distinct test plans. Within each test plan, the main functionalities were structured as test suites, and the corresponding test cases were stored under these test suites in TestLink. Each test case was assigned for execution, and the results were systematically collected across different LLM-generated test plans.

In this study, execution refers to organizing the LLM-generated test cases into suites within TestLink and recording outcomes at the feature level rather than at the unit level or full end-to-end testing. The Testing Scope sections of each generated test plan already include multiple testing levels, such as functional, integration, system, and acceptance, in alignment with ISO/IEC/IEEE 29119-3. This structure demonstrates that although execution was limited to feature-level validation, the generated plans were conceptually designed to support broader testing coverage.

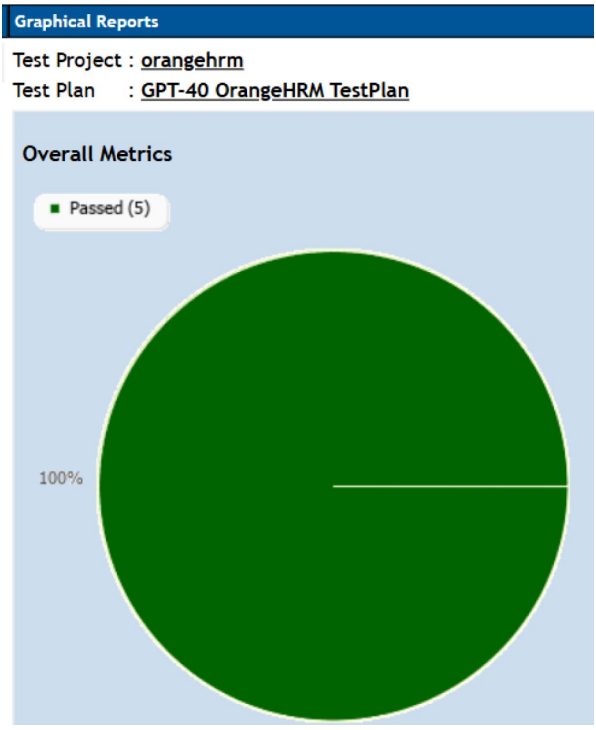
In this study, test case execution outcomes were classified into three categories: Passed, Blocked, and Failed. A test case was marked as Passed when the specified functionality existed in the system and was successfully tested to confirm that the requirement was satisfied. A test case was marked as Blocked when the main functionality might exist in the system, but could not be executed in the current setup due to configuration limitations, licensing restrictions, or other environmental constraints. A test case was marked as Failed when the functionality did not exist in the system at all and was instead generated by the LLM in error, representing a hallucinated functionality.

Test execution was performed manually by the authors. For each test case, the execution status was manually assigned within the TestLink environment based on the observed system behavior.

Figure 7 shows the test execution window of the OrangeHRM application when the GPT-4o-generated test plan was used. The left-hand side of Fig. 7a displays the test suites for Employee Management, Leave Management, Time Tracking, etc. The test cases under each test suite are shown with test case IDs starting with “otc”. Once the execution status of each test case was manually recorded as Passed, TestLink automatically updated the visual indicators and summary metrics, resulting in all test cases being displayed in green in the execu-



(a) Example of TestSuite and the main functionalities listed in the generated test plan.

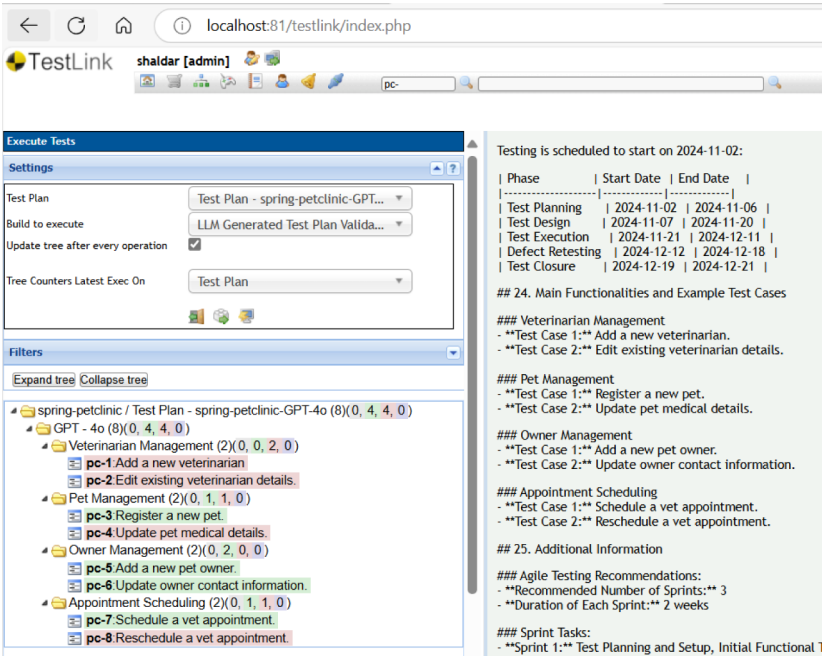


(b) Report generated from the test execution using the GPT-4o suggested main functionalities for the OrangeHRM application.

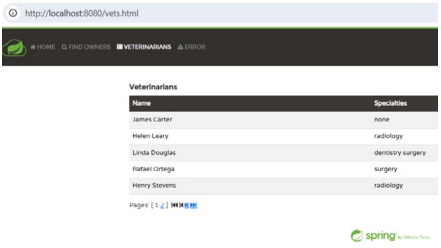
Fig. 7 Comparison of the generated test plan and execution report in TestLink for the OrangeHRM application

tion tree. Figure 7b illustrates one of the artifacts from the report generated from the same test plan using TestLink. This report indicates that out of the five total test cases, all passed, resulting in an entirely green pie chart representing a 100% pass rate in the summary report.

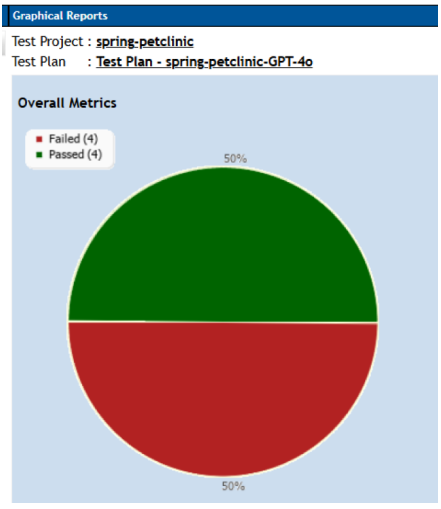
Figure 8 shows the test execution window of the Spring PetClinic application when the GPT-4o-generated test plan was used. The left-hand side of Fig. 8a displays the test suites for Veterinarian Management, Pet Management, Owner Management, and Appointment



(a) Example of TestSuite and the main functionalities listed in the generated test plan.



(b) Report generated from the test execution using GPT-4o suggestions.



(c) Another report showcasing additional test execution details.

Fig. 8 Overview of test suite generation and execution reports in TestLink for the Spring PetClinic application

Scheduling. The test cases under each test suite are shown with test case IDs starting with "PC". After the execution status of each test case was manually recorded, TestLink automatically updated the visual indicators. Failed test cases were displayed in red, while passed

test cases were shown in green. The right-hand side of this figure shows the test plan content generated from GPT-4o, which was pasted into the TestLink test plan. Figure 8b shows why the new veterinarian test case failed testing, as the list of veterinarians does not have any option to add or edit the existing vets. Only the view option is available. Figure 8c illustrates the pie chart generated from TestLink from the test execution result of when these generated test cases from the Pet Clinic were executed. Out of the test cases executed, 4 passed and 4 failed, leaving the pie chart showing 50% in green representing passed testing and the remaining 50% failed testing shown in red.

Table 4 summarizes the execution results, including test pass rates and failure analysis. The results show that for the OrangeHRM application, most models achieved a 100% success rate, except for Gemini, which had a 25% blocked rate. Some test cases were marked as blocked because certain features were either not configured or unavailable in the selected system. For instance, while the 'Forgot Email Address' feature was present, it did not function due to the OrangeHRM application not being configured to receive email notifications. Similarly, some features, such as 'Employee Training' and 'Generating Payroll Reports', were missing from the menu options. The high success rates observed for the OrangeHRM application indicate that, within the scope of this study, systems with clearly defined workflows and stable feature sets may facilitate more accurate identification of core functionalities by LLMs. As OrangeHRM is an actively maintained, real-world application with well-documented modules, the generated test plans showed stronger alignment with executable system behavior across models. However, this observation is limited to the evaluated system and should not be interpreted as a general characteristic of all HR management platforms. Gemini, despite generating the highest number of test cases, encountered blocked cases due to its broader coverage, which included scenarios not supported by the specific system configuration. Models that generated fewer test cases, such as Llama3, Mistral, GPT-4o, and Cohere, demonstrated higher precision by focusing on applicable scenarios.

A notable outcome of this study was that, without prior domain knowledge of these applications, the test lead could gain an understanding of the core functionalities through the generated test plan. These LLM-generated test plans can act as a baseline template to

Table 4 Execution results from TestLink for the generated test plans

Application	Model Name	Test Suites	Passed (P)	Failed (F)	Blocked (B)	Total	P%	F%	B%
OrangeHRM	**Gemini**	**8**	24	0	8	**32**	75	0	25
	Llama3	5	15	0	0	15	100	0	0
	Cohere	3	3	0	0	3	100	0	0
	GPT-4o	5	5	0	0	5	100	0	0
	Mistral	6	12	0	0	12	100	0	0
Spring PetClinic	Gemini	6	8	10	0	**18**	44	56	0
	Llama3	7	2	10	0	12	17	83	0
	Cohere	4	2	6	0	8	25	75	0
	GPT-4o	4	5	5	0	10	50	50	0
	Mistral	4	10	6	0	16	63	38	0
ZenCart	Gemini	5	23	2	0	**25**	92	8	0
	Llama3	2	3	1	0	4	75	25	0
	Cohere	4	5	2	0	7	71	29	0
	GPT-4o	3	3	3	0	6	50	50	0
	Mistral	5	8	2	0	10	80	20	0

Notes. Results compiled from TestLink executions across all generated test plans

quickly familiarize themselves with system functionalities. With human refinement, these generated test plans can serve as an efficient starting point, reducing the time required for test planning and ensuring that fundamental system features are covered. This has strong implications for improving test coverage and onboarding efficiency in software testing, particularly when working with unfamiliar applications.

For Spring PetClinic, the failure rates were significantly higher, with Llama3 experiencing the most failures at 83%. Out of 18 executed test cases, 10 failed, indicating limitations in its generated test plans. Among the five models, Gemini had six different test suites with a total of 18 test cases, the highest among all models for this application, and exhibited a failure rate of 56%, which was relatively lower than Llama3. GPT-4o produced similar results but generated only 10 test cases, fewer than both Gemini and Llama3. Mistral demonstrated better performance, achieving a 63% pass rate. It generated a total of 16 test cases, fewer than Gemini but more than the other models. The higher failure rates observed for Spring PetClinic may be partially attributed to its role as a demonstration-oriented application rather than a production-deployed system. In the evaluated version, several functionalities are intentionally simplified or incomplete, which may have limited the ability of LLMs to accurately infer realistic usage scenarios. This interpretation is based solely on the evaluated application and does not imply that all demonstration systems exhibit similar behavior. Unlike OrangeHRM and ZenCart, which are actively deployed in real-world environments, Spring PetClinic is used mainly as a sample application for showcasing the Spring framework. This means that test cases generated by LLMs may not align well with practical usage scenarios, leading to higher failure rates. Practical usage scenarios in this context refer to real-world functional sequences executed by end-users within the tested application, such as completing transactions, managing workflows, or performing role-specific operations. These represent higher-level behavioral flows rather than isolated unit-level actions.

Additionally, as a Java-based system with dependencies across CSS, Java, HTML, SCSS, and Docker configurations, the overall complexity of the repository may have contributed to the difficulty in generating relevant test cases. In this study, the GitHub repository URL was provided as the primary input, allowing the models to interpret the project context from publicly available metadata, documentation, and repository structure rather than direct code parsing. The availability of diverse file types and configurations within the repository increased contextual complexity, which may have influenced the level of precision in the generated test plans. The models that struggled likely failed to capture the intricate interactions between the user interface and backend, resulting in a higher proportion of failed test cases. Mistral's better performance suggests that it was able to balance coverage and relevance more effectively than the other models, while GPT-4o demonstrated consistent functional accuracy with a moderate 50% pass rate, avoiding the fluctuations seen in Gemini and Llama3.

For the ZenCart application, Gemini's generated test plan showed the best performance, achieving a 92% pass rate with the maximum number of generated test cases at 25. However, GPT-4o and Cohere struggled with 50% and 71% pass rates, respectively. The strong performance of Gemini in this application suggests that e-commerce platforms, like HR systems, have structured workflows, making them more suited for extensive test generation. Since e-commerce applications follow predefined processes for product management, checkout, and order tracking, the larger number of test cases generated by Gemini contributed to its success. The real-world applicability of ZenCart, similar to OrangeHRM, may

have also played a role in improving test generation accuracy, as structured functionalities tend to be better documented and more frequently encountered in model training data. Other models produced fewer test cases, resulting in a lower failure rate; however, this also meant that certain functionalities might have been missed in their test plans. GPT-4o, similar to its performance in Spring PetClinic, maintained a moderate pass rate, indicating that while it did not generate as many test cases as Gemini, the ones it produced were more targeted and applicable.

These findings suggest that different models exhibit strengths and weaknesses depending on the nature of the application under test. Applications with real-world deployments, such as OrangeHRM and ZenCart, benefit from models that generate precise, targeted test cases, as seen with Llama3, Mistral, and GPT-4o. More complex applications, such as Spring PetClinic, require models that can handle UI and backend dependencies effectively, with Mistral showing the best balance between coverage and accuracy. Additionally, the nature of the application, whether it is a real-world system or a demo project, impacts the effectiveness of LLM-generated test cases. Within the limited set of evaluated applications, systems with more stable and well-documented workflows exhibited higher alignment between generated and executable test cases, whereas the demonstration-oriented application showed greater variability. These observations are indicative rather than conclusive and highlight the need for broader empirical validation across a wider range of application types. The frequency of updates and the extent to which an application is actively maintained also influence the accuracy of generated test plans. GPT-4o showed a balanced performance across different applications, maintaining a moderate level of accuracy without significant fluctuations in test pass rates. While it did not produce the highest number of test cases, it maintained consistency across different application types, making it a reliable choice when test stability is prioritized. Furthermore, the ability of LLM-generated test plans to serve as an initial reference point can enhance efficiency for test leads unfamiliar with a particular application, as demonstrated in OrangeHRM. By providing a structured baseline, these test plans can significantly reduce onboarding time and improve test coverage, particularly in domains where manual test planning would otherwise require extensive learning efforts. Selecting the appropriate model depends on the trade-off between broad test coverage and precision, and careful validation of generated test cases is necessary to ensure their applicability. Also, minimum information was provided this time in the prompt for the application, but based on the LLM's pre-trained model, the outcome is satisfactory. Tailoring the prompts and providing or uploading more detailed information about the application will increase the performance of the test plans.

5.3 Effort estimation results

The generated test plan included a dedicated section for estimating testing effort and scheduling. To verify whether these estimated efforts followed a logical trend, the total lines of code (LOC) for each application were calculated, under the assumption that larger codebases would generally demand greater testing coverage than smaller ones.

LOC for each application was measured using an automated script that scanned relevant source files while excluding unnecessary directories and non-source files. For ZenCart and OrangeHRM, both PHP-based, only `.php` files were counted, whereas for Spring PetClinic, which is Java-based, only `.java` files located in the `src/main/java` directory

were considered. To ensure accuracy, directories such as `bin`, `build`, `test`, `cache`, and `config` were excluded from the count, preventing the inclusion of configuration files, logs, or test scripts that do not contribute to development effort. The LOC computation used Python's `mmap` module, which reads files in a memory-mapped manner and efficiently counts newline characters to determine line counts.

To provide additional context beyond LOC, repository-level indicators such as the number of forks, stars, and watchers were analyzed in Table 1. These metrics reflect project maturity, popularity, and community engagement, which often influence documentation quality and model interpretability. Projects with higher activity and visibility tend to have richer contextual information, enabling the LLMs to generate more accurate and comprehensive test artifacts.

The estimated testing effort results for each application and model are summarized in Table 5 and visualized in Fig. 9, which combines effort estimates with corresponding LOC to illustrate how LLM-based effort predictions scale with project size.

The LOC was considered not as a proxy for execution time, but for potential testing scope, since larger or more complex codebases typically require broader test coverage. This interpretation aligns with prior studies where software size and structural measures such as lines of code and cyclomatic complexity were found to correlate with testing effort and comprehension complexity (Kushwaha & Misra, 2008). Furthermore, quantitative estimation frameworks such as qEstimation (Nguyen et al., 2013) reinforce that measurable indi-

Table 5 Schedule or estimation of effort by model and application

Application	LOC	Relative Size	Gemini 1.5	GPT-4o	Llama3	Cohere	Mistral
Spring PetClinic	1 644	Small	56–84	49	60	49	57
ZenCart	37 185	Medium	56	84	42	56	67
OrangeHRM	207 126	Large	70	112	28	86	42

Notes. Estimates correspond to the "Schedule or Estimation of Effort" section in the generated test plans

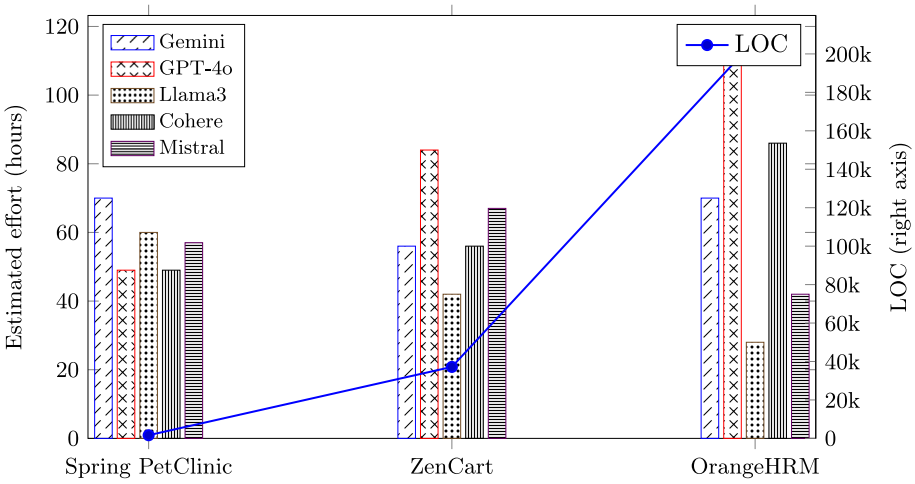


Fig. 9 Estimated testing effort by model (grouped bars, left axis) overlaid with application size in LOC (line, right axis). For Gemini on Spring PetClinic, the prompt returned a range (56–84 hours); the midpoint (70) is plotted for visual comparison

cators of test scope, whether based on LOC, test checkpoints, or repository-level metrics, are essential for approximating testing effort in practical settings.

To complement LOC, additional repository-level indicators such as the number of forks, stars, and watchers were also analyzed (see Table 1). These metrics provide broader contextual insight into project maturity, popularity, and maintenance activity, which can influence both the availability of descriptive resources and the accuracy of LLM-generated artifacts. Higher engagement levels typically correspond to richer documentation and community discussions, thereby improving model interpretability and testing relevance.

The estimated test effort results for each application using different LLM models are presented in Table 5.

A key expectation in effort estimation is that as the size of an application (LOC) increases, the estimated effort should not decrease. However, discrepancies were observed in the LLM-generated estimations. GPT-4o and Gemini produced higher estimates (e.g., 112 hours for OrangeHRM), aligning better with real-world expectations. In contrast, Llama3 provided significantly lower estimates, predicting only 28 hours for OrangeHRM, despite it being the largest application. This inconsistency suggests that Llama3 does not sufficiently account for codebase size when estimating effort. Additionally, Mistral's estimates fluctuated, producing a lower effort estimate (42 hours) for OrangeHRM than for ZenCart (67 hours), contradicting the expected trend. Cohere provided more balanced estimates, though still slightly underestimated for large applications.

These findings indicate that while LLMs show potential for automated test effort estimation, their predictions vary significantly depending on the model. GPT-4o and Gemini demonstrated a more reliable scaling pattern where effort increased with LOC, while Llama3 and Mistral lacked consistency in their estimations. Future refinements in LLM-based estimation models may be needed to better account for application size and complexity.

While this study demonstrates that LLMs are capable of generating test effort estimates, their practical value depends on the reasonableness of those outputs rather than their mere availability. To evaluate this, we examined how estimated effort scaled with measurable project attributes such as lines of code (LOC), repository popularity indicators (Stars, Forks, Watchers), and overall application complexity. Among the evaluated models, GPT-4o and Gemini displayed effort estimates that increased proportionally with application size and repository activity, suggesting partial alignment with realistic testing expectations. In contrast, Llama3 and Mistral produced irregular or lower estimates that did not follow the expected scaling pattern, indicating weaker contextual sensitivity. These mixed results show that while some LLMs can approximate logical effort trends, their quantitative reliability remains inconsistent, underscoring the need for calibration with empirical testing data in future work.

5.4 Readability of the generated test plans

For readability evaluation, we focused on the test plans generated using Prompt 2 (Key Fields). This prompt was selected because it not only provided structural coverage of ISO/IEC/IEEE 29119-3 sections but also specified application functionalities, making it suitable for both execution validation and readability analysis.

The comparative readability outcomes for the generated test plans are summarized in Table 6, which lists the Flesch Reading Ease and Gunning Fog Index Scores per model to illustrate variations in textual complexity across applications and prompt styles.

Table 6 Readability score

Application Name	Application Name	Flesch Reading Ease Score	Flesch Reading Ease Category	Flesch-Kincaid Grade Level Score	Flesch-Kincaid Grade Level Category	Gunning Fog Index Score	Gunning Fog Index Category
GPT-4o	spring-petclinic	32.7	Difficult	12	High School	11.05	Standard
	ZenCart	34.73	Difficult	11.2	High School	10.6	Standard
	OrangeHRM	41.66	Difficult	10.6	High School	11.46	Standard
Cohere	Spring PetClinic	34.32	Difficult	11.4	High School	10.61	Standard
	ZenCart	34.73	Difficult	11.2	High School	10.77	Standard
	OrangeHRM	34.22	Difficult	11.4	High School	10.48	Standard
Google_gemini	Spring Petclinic	31.48	Difficult	12.4	High School	11.05	Standard
	ZenCart	33.1	Difficult	11.8	High School	9.68	Standard
	OrangeHRM	28.3	Very Confusing	11.6	High School	8.65	Easy
Llama3	Spring PetClinic	13.41	Very Confusing	19.4	Graduate Level	17.73	Very Difficult
	ZenCart	17.3	Very Confusing	20	Graduate Level	18.2	Very Difficult
	OrangeHRM	33.44	Difficult	13.8	College Level	12.98	Standard
Mistral	Spring Petclinic	52.87	Difficult	8.4	Middle School	7.79	Easy
	ZenCart	45.42	Difficult	9.2	High School	8.04	Easy
	OrangeHRM	25.66	Very Confusing	12.6	High School	11.28	Standard

Table 7 Test items and scope for each LLM-generated test plan

Model	Test Items	Test Scope
GPT-4o	HR management modules (Employee Information, Leave, Time and Attendance), Recruitment, Performance, Administration & configuration utilities.	Evaluate functionality and performance of all OrangeHRM modules; assess security vulnerabilities, compliance with standards, and usability for accessibility.
Gemini	OrangeHRM web application (frontend and backend), REST APIs for external integrations, and documentation (user and API manuals).	All features and functionalities of OrangeHRM, including user/employee management, recruitment, performance, time tracking, reporting, analytics, and third-party integrations.
Llama3	Features from OrangeHRM GitHub (v7.3.0): Manage Organization, Employees, Recruitment, Time-off, Leaves, Performance, Reports, Dashboards, and Setup.	Covers all planned functionalities in version 7.3.0; excludes deprecated features, customizations, and external integrations.
Cohere	Employee recruitment, time tracking, payroll generation, and performance review components of OrangeHRM.	Covers system behavior under normal and abnormal conditions, focusing on completeness and correctness of HR workflows.
Mistral	UI components, business logic, database interactions, API endpoints, and system integrations.	Functional validation, usability, performance, and security testing (authentication, authorization), along with regression testing.

Notes. Condensed for width. All descriptions are summarized from Prompt 3 outputs for the OrangeHRM application

These results indicate that LLMs tend to produce moderately complex textual structures, mainly due to verbose domain- and application-specific language describing sections such as scope, objectives, levels of testing, schedule, risks, environment, and training requirements, along with the use of testing-domain terminology. The generated content is generally suitable for technical audiences such as test managers and testing professionals, but may require simplification for general readers.

For Spring PetClinic, the Flesch Reading Ease score categorized the test plans generated by GPT-4o, Cohere, Gemini, and Mistral as difficult. This outcome is expected since test plans contain technical terminology that may not be easily understood by individuals without prior experience in software testing. However, test leads and testers, who typically hold at least a college degree, are likely to find the level of complexity appropriate for their work. Llama3, on the other hand, classified these test plans as very confusing, indicating a significantly higher level of difficulty. The Flesch-Kincaid Grade Level metric suggested that most test plans were readable at the high school level, except for Llama3, which required graduate-level comprehension. Mistral generated the most readable test plans, making them accessible at a middle school level. The Gunning Fog Index showed consistency across Cohere, Gemini, and GPT-4o, categorizing their test plans within a standard readability range. In contrast, Llama3 classified its test plans as very difficult, while Mistral rated them as easy. A balanced assessment from Gemini, Cohere, and GPT-4o suggests that the readability of the generated test plans is reasonable, and test leads and testers should be able to interpret them based on their qualifications.

For ZenCart, the readability scores followed a similar pattern to those observed for OrangeHRM. However, Mistral's test plan was slightly more complex in this case, requiring high school-level comprehension according to the Flesch-Kincaid index, whereas it was categorized at the middle school level for Spring PetClinic.

For OrangeHRM, Google Gemini's readability scores showed some inconsistencies. The Flesch Reading Ease score categorized the test plan as very confusing, but the Gunning Fog

Index classified it as easy. Mistral's test plan also fell into the very confusing category based on the Flesch Reading Ease score.

Overall, Llama3 generated the most complex test plans, achieving graduate-level readability scores, while Mistral produced the most readable test plans, ranging from middle school to high school levels. GPT-4o and Cohere remained within standard readability levels, while Gemini's complexity varied across applications. While some test plans were categorized as difficult or very confusing by standard readability metrics, this does not necessarily indicate an issue. Test plans are inherently technical documents, and test leads and testers are expected to have the background necessary to comprehend them. However, significant variations in readability across different models suggest that some LLM-generated test plans may require adjustments to ensure clarity and usability without oversimplifying critical technical details.

5.5 Variability in the generated test plans for test items and test scope

Table 7 presents the test items and test scope section generated using Prompt 3. Although Prompt 3 explicitly specified all sections from the standard in its input and successfully generated content for each, the different models still produced varying outputs in terms of test items and test scope. Despite having a structured input, the generated test plans showed inconsistencies in the level of detail, feature selection, and scope definition across models.

While some models, such as Gemini, included both front-end and back-end components, REST APIs, and documentation, GPT-4o focused primarily on HR management modules and system utilities. Llama3 structured its test plan based on features extracted from the OrangeHRM GitHub repository, whereas Cohere concentrated on behavioral aspects, ensuring coverage of system behavior under normal and abnormal conditions. Mistral, in contrast, emphasized technical components, including UI interactions, business logic, API integrations, and security testing.

These differences indicate that even when LLMs are given a predefined structure, they still interpret the test scope differently, prioritizing different aspects of the system under test. This variability occurs not only across models but also within the same model when identical prompts are executed multiple times. For instance, repeated runs with GPT-4o occasionally emphasized different modules or feature flows, reflecting the stochastic nature of generative decoding. While such diversity can expand test coverage by revealing alternative perspectives, it also introduces challenges for reproducibility and consistency in automated testing workflows. While Prompt 3 attempted to create a more structured test plan, the underlying differences in test scope, granularity, and coverage demonstrate the inherent diversity in model-generated content, reinforcing the need for careful evaluation when using LLMs for automated test plan generation.

As summarized in Table 7, this variation in test scope across models demonstrates that even under identical prompt conditions, different LLMs prioritize distinct functional areas, reflecting adaptive interpretation rather than uniform coverage. For example, GPT-4o emphasized HR-centric modules, Gemini expanded coverage to API and integration layers, and Mistral concentrated on validation and performance testing–illustrating that each model applies its own latent bias in determining testing focus.

6 Discussion

The results of this study demonstrate that prompt design and model selection significantly impact the effectiveness of LLM-generated test plans. By evaluating test plan coverage, execution success, effort estimation, and readability, we gained insights into the strengths and limitations of AI-driven test planning, which addresses the earlier research questions.

6.1 Impact of prompt structure on test plan completeness

One of the most notable findings is that prompt completeness directly influences test plan coverage. Using a Minimal Prompt (Prompt 1) resulted in only 38% coverage of ISO/IEC/IEEE 29119 sections, missing key elements such as risk management, test basis, stakeholders, and testing communications. Prompt 2 also missed the test basis and test scope sections. When structured inputs (Prompt 3: Full List) were provided, coverage improved to 100%, demonstrating that providing explicit information to the LLM leads to more complete test plans. This reflects on the RQ2 of "how well do the generated test plans adhere to standard structures?". It appears that, without specifying the sections (Prompt 3), several of the guided sections of ISO/IEC/IEEE 29119 were not available in Prompt 1 and Prompt 2, but the specified sections were generated in Prompt 2 regardless of their not being available in this standard. Hence, the LLM can allow customization, but being specific is the key to guiding the LLMs in the right direction.

These findings also address RQ1, "Can LLMs generate relevant and complete test plans with limited information," by suggesting that while LLMs can generate relevant test plans based on minimal input, their completeness depends on how well the prompt is structured. This implies that LLM-generated test plans are not inherently incomplete but rather limited by the input constraints. Additionally, the study revealed that when repository-specific data was insufficient, some models introduced speculative content, reinforcing concerns about hallucinated details Waldo and Boussard (2024); Leiser et al. (2024). However, the test plans generated for the OrangeHRM application were quite promising, which can certainly add value in the test plan preparation practice, as long as there are some human oversights, and this test plan is considered for brainstorming purposes.

6.2 Execution accuracy: variability across models

The execution of test cases based on LLM-generated plans revealed substantial differences in model performance. The Mistral model produced the most reliable test plans, with a high pass rate across all applications. Llama3 had the highest test failure rate (83% for Spring PetClinic), suggesting that its generated test cases contain logical inconsistencies or missing steps. GPT-4o and Cohere performed consistently well, with Cohere achieving 100% pass rates in OrangeHRM.

The prompts in this study provided only the application name and GitHub repository link, without explicit instructions to analyze the source code. However, since repositories often contain descriptive content such as README files, configuration notes, and usage documentation, the models might have implicitly drawn on this publicly available context. Enterprise-level applications such as OrangeHRM and ZenCart have richer documentation and community resources, which likely contributed to their higher success rates. In contrast, the demonstration-oriented Spring PetClinic repository contains limited descriptive mate-

rial, reducing the model's ability to align generated functionalities with the actual system. These findings suggest that the availability of ecosystem-level information influences LLM performance even in the absence of explicit code analysis.

In addition to repository-related factors, differences in model capacity may also contribute to the observed variability across LLMs. Larger models typically possess broader internal representations derived from training on more extensive and diverse corpora, which may improve their ability to produce test plans aligned with software structures, documentation patterns, and testing workflows. While some models publicly disclose parameter counts, others do not provide such architectural details. As a result, the influence of model size and internal knowledge could not be isolated in this study and is therefore discussed as a contributing factor rather than a determinative explanation.

The repositories contained diverse file types and configurations, including CSS, HTML, and Docker files, which increased contextual complexity. Although the LLMs were not directly instructed to access or parse any source code, this complexity may have influenced the precision of the generated test plans.

As summarized in Table 5, repository attributes such as Total Lines of Code (LOC) were examined to assess project activity and maintenance frequency. LOC serves as an indicator of project size and complexity, helping to contextualize variations in LLM performance across repositories.

Repositories that undergo frequent updates may modify API endpoints or user interface structures, which can lead LLMs relying on publicly available snapshots to generate outdated or inconsistent test plans. Consequently, applications with more stable repositories tend to yield a higher alignment between generated and actual test functionalities.

RQ3: Are test plans generated by different LLMs consistent, readable, and practically applicable when organized and executed in a test management tool such as TestLink?

Execution in this context refers to the manual testing of the system under test (SUT) based on LLM-generated test plans. The generated test plans were first uploaded as textual test planning artifacts and then translated into logical test suites and test cases within TestLink. This process emphasizes feature-level validation rather than unit-level or fully automated end-to-end testing.

From a usability perspective, the readability of the generated test plans was evaluated at the document level using established readability metrics, including the Flesch Reading Ease Score, Flesch–Kincaid Grade Level, and Gunning Fog Index. These metrics were used to assess whether the generated test plans could be understood by test leads and testers with varying levels of experience, independent of individual test case execution outcomes.

During manual test execution, the majority of the generated test cases were readable, well-structured, and could be followed without difficulty. However, several test cases failed or were marked as blocked because certain generated functionalities were not present in the application or did not behave as described. These inconsistencies occurred primarily when application-level requirements were unclear and external contextual input was intentionally limited. In such cases, the LLMs relied on generalized prior knowledge, leading to assumptions that did not fully align with the actual system behavior.

These observations indicate that practical applicability is influenced not only by the selected LLM model but also by prompt structure, the availability of pre-trained knowledge, and the temporal scope of the model's training data. Since large language models are trained

with fixed knowledge cutoffs, recently introduced features, interface changes, or bug fixes may not be reflected in the generated test plans.

Readability scores varied across different LLMs, reflecting differences in language style, level of detail, and document organization; however, the evaluation focused on comparative trends rather than absolute readability thresholds. Overall, while LLM-generated test plans can effectively support early-stage test planning and documentation, human validation remains essential to assess execution feasibility and ensure alignment with evolving software systems.

6.3 Effort estimation: can AI predict testing workload?

Comparing LLM-based effort estimation across different models revealed inconsistencies in predicted testing workload. GPT-4o and Gemini provided more comprehensive estimates, aligning closely with detailed test plans, making them potentially more suitable for project planning. In contrast, Llama3 consistently underestimated effort, predicting significantly lower time requirements for large applications such as OrangeHRM. Mistral and Cohere produced moderate estimates, balancing between optimistic and conservative projections.

These results suggest that while LLMs can assist in effort estimation, their reliability varies depending on the granularity of the generated test plan. Models that generate more concise, high-level test plans like Llama3 tend to underestimate effort, while those producing more detailed and structured test plans (GPT-4o, Gemini) generate more realistic effort estimates. This highlights the importance of validating AI-driven effort predictions before incorporating them into project planning and resource allocation.

For practitioners, this means that AI-driven effort estimation should be used as a guiding tool rather than a definitive calculation. Further refinements, such as fine-tuning models on historical effort data, may improve accuracy.

6.4 Readability

Readability analysis showed that some models produce more comprehensible test plans than others. Mistral generated the most readable test plans (Middle to High School level), making them more accessible for testers of varying experience levels. Llama3 produced highly complex outputs (Graduate-level readability), which might be useful for detailed documentation but could pose usability challenges. GPT-4o and Cohere maintained readability within standard levels, balancing detail and accessibility.

6.5 Practical implications for AI-driven test planning

The findings from this study have several practical applications. Test teams should carefully design prompts to maximize test plan completeness. AI-driven Effort estimation is promising but not yet fully reliable–manual validation is still needed. Readability must be considered when adopting AI-generated test plans to ensure usability across diverse testing

teams. Future research can explore hybrid approaches, such as combining multiple LLM outputs or incorporating human-in-the-loop AI workflows for more reliable test generation.

7 Threat to validity

The quality of the generated test plans depends on how the prompts are structured. While we tested three prompt variations, there may be better prompt designs that could yield superior results. In addition, while the framework allows for entering the description of the application, we kept the scope limited to entering the name of the application and the GitHub link with the assumption that the vast amount of pre-trained LLM models will be able to retrieve information based on existing information. In addition, when only a GitHub repository link is provided, the quality of the generated test plans may be influenced by the extent of knowledge encoded during model pretraining, which is often correlated with the size and capacity of the LLM. Although the generated test plans are showing promising results, extending the scope by including more detailed information may improve the generated test plans.

LLM-based estimates might work differently across software domains. For this work, the generated test plans were validated by providing a minimum set of information for the application. However, providing more detailed instructions or being able to upload the prospective changes for enhancement projects, which generally require regression testing, will benefit from automated test plan generation using LLM.

Results are based on GPT-4o, Mistral, Llama3, Cohere, and Gemini. Future improvements in LLM architecture may alter outcomes. The study was conducted using OrangeHRM, ZenCart, and Spring PetClinic. Results may differ for larger, more complex enterprise applications.

LLM-generated plans for other domains may require different evaluation metrics. While ISO/IEC/IEEE 29119 coverage is a strong indicator, it can increase the reliability if each of the sections generated test plans can be compared to a completed testing document. The readability scores are based on the Flesch-Kincaid, Gunning Fog Index, etc., which may apply to the readability of regular text documents and do not consider technical clarity. The overall readability still indicates whether the test leads would be able to understand the document, so this still serves the purpose.

Additionally, since LLM outputs are non-deterministic, the generated test plans may vary across repeated runs using the same prompt. This output variability can affect reproducibility, introducing concerns related to the internal validity of the model's consistency of behavior. To mitigate this, standardized API parameters such as `temperature` and `max_tokens` were consistently applied across all models to maintain uniform sampling behavior. Furthermore, since the prompts were applied uniformly across models, external validity is supported by the consistency of evaluation criteria between LLMs. To further enhance reproducibility and transparency, all LLM-generated test plans and supporting prompt generation scripts are publicly available at Figshare (https://figshare.com/s/243347265e51fabed262). Still, human oversight remains essential to validate and refine the generated test plans before reliable use, ensuring quality assurance and accountability in AI-assisted test planning.

8 Conclusion and future work

This study demonstrates that large language models (LLMs) can support early-stage software test planning and automated test plan generation, within the scope of the three evaluated applications. In particular, the models were able to generate structured test plans even when only limited application-level information was provided, offering support for early-stage planning decisions such as scope identification, feature prioritization, risk consideration, and structured testing artifact creation.

The results indicate that prompt design plays a critical role in determining test plan quality. In particular, explicitly specifying standard test plan sections within the prompt led to more structured and complete LLM-generated test plans across the evaluated models. This finding highlights the importance of structured prompting when applying LLMs to higher-level software testing activities.

At the same time, variability was observed among different LLMs in terms of test coverage, level of detail, effort estimation, and readability. These differences suggest that model selection influences the characteristics of the generated test plans, even when identical prompts are used. Consequently, LLM-based test plan generation should be viewed as a decision-support mechanism rather than a fully autonomous replacement for expert judgment. In particular, differences in readability indicate that while some models generate test plans that are easier for practitioners to interpret and reuse, others produce more complex documentation that may require additional refinement before adoption.

Human oversight remains essential to review, validate, and refine the generated test plans before practical adoption. While LLMs can assist in reducing documentation overhead and accelerating early-stage test planning, expert involvement is required to ensure contextual correctness, feasibility, and alignment with organizational testing practices.

Future work will extend this investigation to a broader set of applications and domains to better assess the generalizability of the findings. Additional directions include exploring hybrid approaches that combine outputs from multiple LLMs to improve completeness, incorporating human-in-the-loop validation mechanisms where test leads iteratively refine generated test plans, and examining how providing richer application artifacts or domain-specific context influences test plan quality, coverage, and consistency.

Acknowledgements The authors would like to acknowledge Dr. Dev Sainani, the Associate Dean of the School of Information Technology, Fanshawe College, for supporting this work.

Author Contributions S. H. conceptualized and designed the research approach, implemented the methodology, conducted the experiments, drafted the initial manuscript, and revised it to produce the final version. L.F.C. supervised the research project, provided critical feedback to refine the methodology, validated the experimental results and supplied the necessary resources for the research. Both authors reviewed and approved the submitted manuscript.

Funding No funding was obtained for this work.

Data Availability The generated test plans using the three different prompts utilizing all 5 models can be found from the following FigShare link: https://figshare.com/s/243347265e51fabed262.

Declarations

Conflicts of interest/Competing interests The authors declare no competing interests.

Competing interests The authors declare no competing interests.

References

Alagarsamy, S., Tantithamthavorn, C., Takerngsaksiri, W., Arora, C., & Aleti, A. (2025). Enhancing large language models for text-to-testcase generation. *Journal of Systems and Software,230*, 112531. https://doi.org/10.1016/j.jss.2025.112531. Retrieved from https://www.sciencedirect.com/science/article/pii/S0164121225001992

Apache Software Foundation (n.d.). Apache jmeter (performance testing tool). Available at: https://jmeter.apache.org/.

Appium Contributors (2025). Appium documentation. Retrieved from https://appium.io/docs/en/latest/ (Accessed: 2026–01-26).

Atlassian (2025). Jira software. Retrieved from https://www.atlassian.com/software/jira (Accessed: 2025–02-06).

Bach, J., Bertolino, A., Kaner, C., Glass, R. L., & Collard, R. (2006). Software Testing and Industry Needs. *IEEE Software, 23*(04), 55–57. https://doi.org/10.1109/MS.2006.113

Balsam, S., & Mishra, D. (2025). Web application testing-challenges and opportunities. *Journal of Systems and Software,219*, 112186. https://doi.org/10.1016/j.jss.2024.112186https://www.sciencedirect.com/science/article/pii/S0164121224002309

Belzner, L., Gabor, T., & Wirsing, M. (2024). Large language model assisted software engineering: Prospects, challenges, and a case study. B. Steffen (Ed.), Bridging the gap between ai and reality (pp. 355–374). Cham: Springer Nature Switzerland.

Beyoğlu, M. M., Kaya, E., & Karabulut, E. (2024). Assessment of the quality, readability, and usefulness of chatgpt generated medical information for ten common cancer types. *Universal Access in the Information Society,* pp. 1–7. https://doi.org/10.1007/s10209-024-01155-6.

Bozic, J., & Wotawa, F. (2019). Software testing: According to plan! 2019 ieee international conference on software testing, verification and validation workshops (icstw) (p. 23–31).

Chan, C. K. Y., & Tsi, L. H. (2024). Will generative ai replace teachers in higher education? a study of teacher and student perceptions. *Studies in Educational Evaluation,83*, 101395. https://doi.org/10.1016/j.stueduc.2024.101395, Retrieved from https://www.sciencedirect.com/science/article/pii/S0191491X24000749.

Cocci, A., Pezzoli, M., Re, M. L., et al. (2024). Quality of information and appropriateness of chatgpt outputs for urology patients. *Prostate Cancer and Prostatic Diseases, 27*, 103–108. https://doi.org/10.1038/s41391-023-00705-y

Cohere AI (2025). Cohere: The all-in-one platform for private and secure ai. https://cohere.com/. (Cohere provides cutting-edge multilingual models, advanced retrieval, and an AI workspace tailored for modern enterprises. Accessed: 2025–02-07).

Cypress.io (2025). Test. automate. accelerate. Retrieved from https://www.cypress.io/ (Accessed: 2025–02-06).

Dakhel, A. M., Nikanjam, A., Majdinasab, V., Khomh, F., & Desmarais, M. C. (2024). Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology,171*, 107468. https://doi.org/10.1016/j.infsof.2024.107468, Retrieved from https://www.sciencedirect.com/science/article/pii/S0950584924000739.

Ferreira, M., Viegas, L., Faria, J.P., Lima, B. (2025). Acceptance Test Generation with Large Language Models: An Industrial Case Study. 2025 ieee/acm international conference on automation of software test (ast) (p. 1–11). Los Alamitos, CA, USA: IEEE Computer Society. Retrieved from https://doi.ieeecomputersociety.org/10.1109/AST66626.2025.00007.

Flesch, R. (1948). A new readability yardstick. *Journal of Applied Psychology, 32*(3), 221–233. https://doi.org/10.1037/h0057532

Forgács, I., & Kovács, A. (2024). Modern software testing techniques: A practical guide for developers and testers (1st ed.). Berkeley, CA: Apress. Retrieved from https://doi.org/10.1007/978-1-4842-9893-0 (Softcover ISBN: 978-1-4842-9892-3, eBook ISBN: 978-1-4842-9893-0).

Gallegos, I. O., Rossi, R. A., Barrow, J., Tanjim, M. M., Kim, S., Dernoncourt, F., & Ahmed, N. K. (2024). Bias and fairness in large language models: A survey. *Computational Linguistics,50*(3), 1097–1179. https://doi.org/10.1162/coli_a_00524. Retrieved from https://aclanthology.org/2024.cl-3.8/

Garousi, V., Felderer, M., Kuhrmann, M., Herkiloğlu, K., Eldh, S. (2020). Exploring the industry's challenges in software testing: An empirical study. *J. Softw. Evol. Process,32*(8). Retrieved from https://doi.org/10.1002/smr.2251.

Grigera, J., Garrido, A., Panach, J. I., Distante, D., & Rossi, G. (2016). Assessing refactorings for usability in e-commerce applications. *Empirical Software Engineering, 21*(3), 1224–1271. https://doi.org/10.1007/s10664-015-9384-6

Groq, Inc. (2025). Groq API Documentation. Retrieved from https://console.groq.com/docs/api-reference. (Accessed: 2025–02-08).

Gunning, R. (1952). *The technique of clear writing*. McGraw-Hill.

Gurock Software GmbH (2024). Testrail (test case management tool). https://www.gurock.com/testrail/. (Accessed: 2024–10-01).

Heusser, M., & Larsen, M. (2023). Software testing strategies: A testing guide for the 2020s. Packt Publishing Ltd.

Homès, B. (2012). *Fundamentals of software testing*. London, UK and Hoboken, NJ, USA: Wiley-ISTE.

IEEE Computer Society (2024). Guide to the software engineering body of knowledge (swebok) (4.0 ed.). Los Alamitos, CA, USA: IEEE Computer Society. Retrieved from https://www.computer.org/education/bodies-of-knowledge/software-engineering (SWEBOK 4.0 reflects modern practices in software engineering, including Agile, DevOps, and continuous testing.).

IEEE Standards Committee (1998). Ieee standard for software test documentation (Tech. Rep. No. IEEE 829–1998). IEEE. Retrieved from https://standards.ieee.org/standard/829-1998.html.

Institute of Electrical and Electronics Engineers (2008). IEEE Standard for Software and System Test Documentation (Tech. Rep. No. IEEE 829–2008). New York, NY, USA: IEEE. Retrieved from https://ieeexplore.ieee.org/document/4601587.

Institute of Electrical and Electronics Engineers (2013). ISO/IEC/IEEE International Standard 29119 Software Testing (Tech. Rep. No. ISO/IEC/IEEE 29119–1 to 29119–5). New York, NY, USA: IEEE Computer Society. Retrieved from https://ieeexplore.ieee.org/document/6506911.

International Organization for Standardization, International Electrotechnical Commission, IEEE. (2021). ISO/IEC/IEEE 29119–3:2021 - software and systems engineering - software testing - part 3: Test documentation. https://www.iso.org/standard/72255.html. (Second Edition, Accessed: 2025–06-07).

International Software Testing Qualifications Board (2024a). Certified tester advanced level test management syllabus. Retrieved from https://istqb.org. (Accessed: 2026–01-27).

International Software Testing Qualifications Board (2024b). https://www.istqb.org/wp-content/uploads/2024/11/ISTQB_CTFL_Syllabus_v4.0.1.pdf (Accessed: 2025–02-28).

International Software Testing Qualifications Board (2025). Certified tester specialist level syllabus – testing with generative ai (ct-genai), v1.0 (Tech. Rep.). ISTQB. Retrieved from https://www.istqb.org/ (Version 1.0, Published 25 July 2025).

Karmarkar, H., Agrawal, S., Chauhan, A., & Shete, P. (2024). Navigating confidentiality in test automation: A case study in llm driven test data generation. 2024 ieee international conference on software analysis, evolution and reengineering (saner) (p. 337–348).

Karpurapu, S., Myneni, S., Nettur, U., Gajja, L. S., Burke, D., Stiehm, T., & Payne, J. (2024). Comprehensive evaluation and insights into the use of large language models in the automation of behavior-driven development acceptance test formulation. *IEEE Access, 12*, 58715–58721. https://doi.org/10.1109/ACCESS.2024.3391815

Katalon (2025). Katalon is your all-in-one test automation solution. Retrieved from https://katalon.com/ (Accessed: 2025–02-06).

Kincaid, J.P., Fishburne, R.P.J., Rogers, R.L., & Chissom, B.S. (1975). Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel (Tech. Rep. Nos. Research Branch Report 8–75). Millington, TN, USA: Naval Technical Training Command, Research Branch. Retrieved from https://stars.library.ucf.edu/istlibrary/56.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). Large language models are zero-shot reasoners. *Advances in neural information processing systems, 35*, 22199–22213.

Kushwaha, D. S., & Misra, A. K. (2008). Software test effort estimation. *ACM SIGSOFT Software Engineering Notes, 33*(3), 1–5. https://doi.org/10.1145/1360602.1361211

Leiser, F., Eckhardt, S., Leuthe, V., Knaeble, M., Mädche, A., Schwabe, G., & Sunyaev, A. (2024). Hill: A hallucination identifier for large language models. Proceedings of the 2024 chi conference on human factors in computing systems. New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3613904.3642428.

Leotta, M., Paparella, D., & Ricca, F. (2024). Mutta: a novel tool for e2e web mutation testing. *Software Quality Journal, 32*(1), 5–26. https://doi.org/10.1007/s11219-023-09616-6

Li, T., Cui, C., Huang, R., Towey, D., & Ma, L. (2025). Large language models for automated web-form-test generation: An empirical study. *ACM Trans. Softw. Eng. Methodol.* https://doi.org/10.1145/3735553.

Li, Y., Liu, P., Wang, H., Chu, J., & Wong, W. E. (2025). Evaluating large language models for software testing. *Computer Standards & Interfaces,93*, 103942. https://doi.org/10.1016/j.csi.2024.103942, Retrieved from https://www.sciencedirect.com/science/article/pii/S0920548924001119.

Ling, Y., Yu, S., Fang, C., Pan, G., Wang, J., & Liu, J. (2025). Redefining crowdsourced test report prioritization: An innovative approach with large language model. *Information and Software Technology,179*, 107629. https://doi.org/10.1016/j.infsof.2024.107629, Retrieved from https://www.sciencedirect.com/science/article/pii/S0950584924002349.

Ma, H., Zhang, C., Bian, Y., Liu, L., Zhang, Z., Zhao, P., & Wu, B. (2023). Fairness-guided few-shot prompting for large language models. *Advances in Neural Information Processing Systems, 36*, 43136–43155.

Marchetto, A., Tiella, R., Tonella, P., Alshahwan, N., & Harman, M. (2011). Crawlability metrics for automated web testing. *International Journal on Software Tools for Technology Transfer, 13*(2), 131–149. https://doi.org/10.1007/s10009-010-0177-3

Micro Focus (n.d.). Hp alm (application lifecycle management). Available at: https://www.microfocus.com/en-us/products/application-lifecycle-management/overview

Milchevski, D., Frank, G., Hätty, A., Wang, B., Zhou, X., & Feng, Z. (2025). Multi-step generation of test specifications using large language models for system-level requirements. G. Rehm and Y. Li (Eds.), Proceedings of the 63rd annual meeting of the association for computational linguistics (volume 6: Industry track) (pp. 132–146). Vienna, Austria: Association for Computational Linguistics. Retrieved from https://aclanthology.org/2025.acl-industry.11/.

Mistral AI (2024). Mistral ai - frontier ai in your hands. Retrieved from https://mistral.ai/en (Accessed: 2025–02-08).

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing* (3rd ed.). Hoboken, NJ: John Wiley & Sons.

Nguyen, H.Q., Johnson, B., Hackett, M. (2003). Testing applications on the web: Test planning for mobile and internet-based systems (2nd ed.). Wiley.

Nguyen, V., Pham, V., & Lam, V. (2013). qestimation: a process for estimating size and effort of software testing. In *Proceedings of the 2013 international conference on software and system process (icssp '13),* (pp. 20–28). Association for Computing Machinery.

Nidagundi, P., & Novickis, L. (2017). New method for mobile application testing using lean canvas to improving the test strategy. In *2017 12th international scientific and technical conference on computer sciences and information technologies (csit),* (Vol. 1, p. 171–174).

OpenAI (2024a). https://platform.openai.com/docs/ (Accessed: 2024–11-12).

OpenAI (2024b). Retrieved from https://platform.openai.com/docs/models#gpt-4o (Accessed: 2025–02-08).

OrangeHRM Team (2025). OrangeHRM: Human Resource Management System. Retrieved from https://github.com/orangehrm/orangehrm (Accessed: 2025–02-06).

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C.L., Mishkin, P. & Lowe, R. (2022). Training language models to follow instructions with human feedback. In *Proceedings of the 36th international conference on neural information processing systems*. Red Hook, NY, USA: Curran Associates Inc.

Paessler AG (n.d.). Paessler web server stress tool. https://www.paessler.com/tools/webstress. (Accessed: 2024–10-01).

Patel, S., & Shah, V. (2015). Automated testing of software-as-a-service configurations using a variability language. In *Proceedings of the 19th international conference on software product line,* (p. 253–262). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2791060.2791072.

Pichai, S., & Hassabis, D. (2024). Our next-generation model: Gemini 1.5. https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/. (Accessed: 2025–10-17).

Pornprasit, C., & Tantithamthavorn, C. (2024). Fine-tuning and prompt engineering for large language models-based code review automation. *Information and Software Technology, 175*, Article 107523.

Prompt Engineering Guide (2025). Prompt engineering guide. https://www.promptingguide.ai/. (Accessed: 2025–02-22).

Saboor Yaraghi, A., Holden, D., Kahani, N., & Briand, L. (2025). Automated test case repair using language models. *IEEE Transactions on Software Engineering, 51*(4), 1104–1133. https://doi.org/10.1109/TSE.2025.3541166

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),* (p. 353–360).

Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering, 50*(1), 85–105. https://doi.org/10.1109/TSE.2023.3334955

Schulhoff, S., Ilie, M., Balepur, N., Kahadze, K., Liu, A., Si, C. others (2024). The prompt report: a systematic survey of prompt engineering techniques. arXiv:2406.06608.

Schur, M., Roth, A., & Zeller, A. (2015). Mining workflow models from web applications. *IEEE Transactions on Software Engineering, 41*(12), 1184–1201. https://doi.org/10.1109/TSE.2015.2461542

Selenium Project (2024). Selenium (web browser automation tool) (Tech. Rep.). SeleniumHQ. Retrieved from https://www.selenium.dev/ (Accessed: 2024–10-01).

Setiani, N., Ferdiana, R., & Hartanto, R. (2020). Test case understandability model. IEEE. *Access, 8*, 169036–169046. https://doi.org/10.1109/ACCESS.2020.3022876

Spring Team (2025). Spring PetClinic. Retrieved from https://github.com/spring-projects/spring-petclinic (Accessed: 2025–02-06).

Strandberg, P.E., Frasheri, M., & Enoiu, E.P. (2021). Ethical ai-powered regression test selection. In *2021 ieee international conference on artificial intelligence testing (aitest),* (p. 83–84).

Tang, Y., Liu, Z., Zhou, Z., & Luo, X. (2024). Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering, 50*(6), 1340–1359. https://doi.org/10.1109/TSE.2024.3382365

TestLink Community (2025). TestLink: Open Source Test Management. Retrieved from https://testlink.org/ (Accessed: 2025–02-08).

Tip, F., Bell, J., & Schäfer, M. (2025). Llmorpheus: Mutation testing using large language models. *IEEE Transactions on Software Engineering, 51*(6), 1645–1665. https://doi.org/10.1109/TSE.2025.3562025

Waldo, J., & Boussard, S. (2024). Gpts and hallucination: Why do large language models hallucinate? *Queue, 22*(4), 19–33. https://doi.org/10.1145/3688007

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering, 50*(4), 911–936. https://doi.org/10.1109/TSE.2024.3368208

Wang, W., Yang, C., Wang, Z., Huang, Y., Chu, Z., Song, D., & Ma, L. (2025). Testeval: Benchmarking large language models for test case generation. Findings of the association for computational linguistics: Naacl 2025 (pp. 3547–3562).

Wang, Y., Zhong, W., Huang, Y., Shi, E., Yang, M., Chen, J., & Zheng, Z. (2025). Agents in software engineering: Survey, landscape, and vision. *Automated Software Engineering, 32*(2), 70. https://doi.org/10.1007/s10515-025-00544-2

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems, 35*, 24824–24837.

Winkler, D., Urbanke, P., & Ramler, R. (2024). Investigating the readability of test code. *Empirical Software Engineering, 29*(2), 53. https://doi.org/10.1007/s10664-023-10390-z

Winteringham, M. (2024). Software testing with generative ai. Shelter Island, NY: Manning. (ISBN: 9781633437160).

Xu, D., Xu, W., Tu, M., Shen, N., Chu, W., & Chang, C.-H. (2016). Automated integration testing using logical contracts. *IEEE Transactions on Reliability,65*(3), 1205–1222. https://doi.org/10.1109/TR.2015.2494685.

Yang, J., Jin, H., Tang, R., Han, X., Feng, Q., Jiang, H., & Hu, X. (2024). Harnessing the power of llms in practice: A survey on chatgpt and beyond. *ACM Trans. Knowl. Discov. Data, 18*(6), 1–32. https://doi.org/10.1145/3649506

Yeh, H.- W., Ma, S.- P., & Chen, Y. (2024). Test case migration from monolith to microservices using large language models. In *2024 ieee international conference on e-business engineering (icebe),* (p. 29–35).

Yeow, J.S., Rana, M.E., & Abdul Majid, N.A. (2024). An automated model of software requirement engineering using gpt-3.5. In *2024 asu international conference in emerging technologies for sustainability and intelligent systems (icetsis),* (p. 1746–1755).

ZenCart Developers (2025). ZenCart: E-commerce Platform. Retrieved from https://github.com/zencart/zencart (Accessed: 2025–02-06).

Zhang, Q., Sun, W., Fang, C., Yu, B., Li, H., Yan, M., & Chen, Z. (2025). Exploring automated assertion generation via large language models. *ACM Trans. Softw. Eng. Methodol.,34*(3). Retrieved from https://doi.org/10.1145/3699598.

Zhao, P., Zhang, H., Yu, Q., Wang, Z., Geng, Y., Fu, F., & Cui, B. (2026). Retrieval-augmented generation for ai-generated content: A survey. Data Science and Engineering, 1–29.

Zhou, S., Jeong, H., & Green, P. A. (2017). How consistent are the best-known readability equations in estimating the readability of design standards? *IEEE Transactions on Professional Communication, 60*(1), 97–111.

Susmita Haldar received her bachelor's and master's degrees in Computer Science from Concordia University, Montreal, Canada, followed by a Ph.D. degree from the Department of Electrical and Computer Engineering at Western University, London, Canada. She worked for several years in the industry in the domains of software testing, application development, business systems analysis, and project management. Since 2019, she has been a Professor at the School of Information Technology, Fanshawe College, London, Ontario, Canada. She currently serves as the Program Coordinator and Professor of the Post-Graduate Certificate Program in Software and Information Systems Testing at Fanshawe College. Her research interests include software testing education, the Scholarship of Teaching and Learning (SoTL), large language models, and the application of machine learning and artificial intelligence in software testing.

Luiz Fernando Capretz has vast experience in the software engineering field as a practitioner, manager, and educator. Before joining Western University (Canada), he worked at both technical and managerial levels, taught, and did research on the engineering of software in Brazil, Argentina, England, Japan, the UAE, Malaysia, and Singapore. He is currently a full professor of software engineering and the director of the software engineering program. He has held visiting positions at the University of Sharjah, New York University in Abu Dhabi, Universiti Teknologi PETRONAS, Yale-NUS College, and at the National University of Singapore (NUSC). His research interests span the fields of software engineering, human aspects of software engineering, software testing, and Generative AI for software engineering. He can be reached at lcapretz@uwo.ca; for further information, please visit http://eng.uwo.ca/electrical/faculty/capretz_l.