

Acceptance Test Generation with Large Language Models: An Industrial Case Study

Abstract—Large language model (LLM)-powered assistants are increasingly used for generating program code and unit tests, but their application in acceptance testing remains underexplored. To help address this gap, this paper explores the use of LLMs for generating executable acceptance tests for web applications through a two-step process: (i) generating acceptance test scenarios in natural language (in Gherkin) from user stories, and (ii) converting these scenarios into executable test scripts (in Cypress), knowing the HTML code of the pages under test. This two-step approach supports acceptance test-driven development, enhances tester control, and improves test quality. The two steps were implemented in the AutoUAT and Test Flow tools, respectively, powered by GPT-4 Turbo, and integrated into a partner company’s workflow and evaluated on real-world projects. The users found the acceptance test scenarios generated by AutoUAT helpful 95% of the time, even revealing previously overlooked cases. Regarding Test Flow, 92% of the acceptance test cases generated by Test Flow were considered helpful: 60% were usable as generated, 8% required minor fixes, and 24% needed to be regenerated with additional inputs; the remaining 8% were discarded due to major issues. These results suggest that LLMs can, in fact, help improve the acceptance test process, with appropriate tooling and supervision.

Index Terms—Acceptance Testing, Large Language Models, Automatic Test Generation, Web Application Testing

I. INTRODUCTION

Recent advances in generative artificial intelligence (AI) and large language models (LLMs) [1] have enabled AI-powered assistants, like GitHub Copilot¹, capable of accurately generating code and unit tests from textual prompts [2], enhancing developer and tester productivity [3]–[5]. However, LLMs are still underexplored for acceptance test (AT) generation [6]—the highest test level, focused on validating that the software meets user expectations and business requirements [7].

To help address this gap, this paper explores the use of LLMs to improve the acceptance test process, especially in the context of agile processes and practices, such as: (i) capturing user requirements as user stories (US); (ii) refining them with (user) acceptance tests written in a restricted natural language (NL) accessible for non-technical users and amenable for subsequent automation, like Gherkin² [8], following acceptance test-driven development (ATDD) [9] and behaviour-driven development (BDD) [10] principles; (iii) automating them using popular frameworks like Selenium³ or Cypress⁴ [11], to reduce manual testing [12] and enable continuous integration (CI).

¹<https://github.com/features/copilot>

²<https://cucumber.io/docs/gherkin/>

³<https://www.selenium.dev/>

⁴<https://www.cypress.io/>

More specifically, we aim to partially automate the last two activities by leveraging LLMs’ natural language understanding and code generation capabilities to: (i) automatically generate acceptance test scenarios from user stories, and (ii) automatically convert these scenarios into executable test scripts.

Our research addresses two key questions:

- **RQ1:** To what extent can LLMs help in generating accurate and comprehensive acceptance test scenarios in natural language from user stories?
- **RQ2:** To what extent can LLMs help in generating accurate and complete⁵ executable test scripts from acceptance test scenarios in natural language?

To address them, we implemented the two generation steps in separate tools—AutoUAT and Test Flow—which can be used independently or together. To ease integration and evaluation in a partner company in the automotive sector, we used GPT-4 Turbo as the LLM, Gherkin and TypeScript as the target languages, and Cypress as the test automation framework.

While assessing long-term productivity and quality gains would require further studies, we expect to help address common industry issues: (i) AT writing is often seen as tedious and low-priority, risking missed edge cases; and (ii) test script writing adds workload for developers, potentially shifting focus from understanding requirements to test success [12].

The main contributions of our work are:

- **AutoUAT** - an LLM-powered tool, integrated in an industrial environment, that automatically generates acceptance test scenarios in Gherkin from user stories;
- **Test Flow** - an LLM-powered tool, integrated in an industrial environment, that automatically generates executable test scripts for end-to-end web application testing in Cypress, based on Gherkin test scenarios’ specifications and the HTML code of the web pages under test purged from irrelevant information, with minimal user intervention;
- **Evaluation results** - results of evaluation studies in our automotive partner, with 95% of acceptance test scenarios generated by AutoUAT and 92% of acceptance test scripts generated by Test Flow considered helpful, and very positive user feedback.

The rest of the paper is organized as follows: Sec. II describes our test generation approach. Sec. III outlines the tool implementation and integration in an industrial workflow. Sec. IV presents experimental results and user feedback, addressing our research questions. Sec. V discusses related work, and Sec. VI concludes with key findings and future directions.

⁵By ‘complete’, we mean covering all steps.

II. ACCEPTANCE TEST GENERATION APPROACH

A. Overall Approach

Our two-step approach for generating executable acceptance test scripts from user stories is outlined in Fig. 1.

The first LLM-powered tool — AutoUAT — was designed to generate Gherkin acceptance test scenarios directly from user story titles and descriptions. The second LLM-powered tool — TestFlow — then takes these Gherkin test scenarios as input, along with the initial user story description and the HTML code of the web pages under test, to generate executable test scripts in Cypress (in TypeScript syntax).

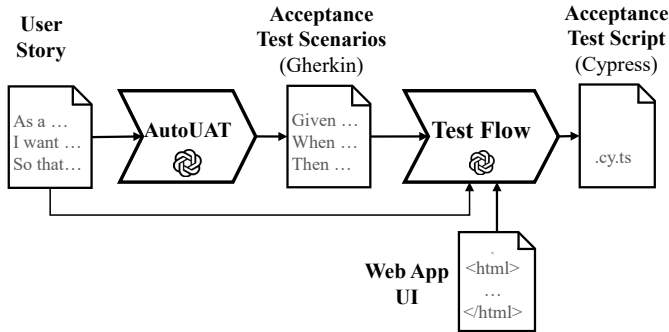


Fig. 1. Acceptance Test Generation with AutoUAT and Test Flow.

This approach was designed to effectively support ATDD: NL test scenarios can be generated when new user stories are created to clarify requirements and guide implementation; once the application user interface (UI) is ready, executable test scripts can be created, mapping test steps in the test scenarios to specific actions and UI elements. It also gives testers and other stakeholders more control, as they can review the generated test scenarios and create their own before converting them to executable test scripts, improving test quality.

Model selection and prompt engineering are described in the next subsections.

B. Model Selection

The study required deploying the model in our industrial partner’s Azure environment for privacy and security reasons. Thus, we evaluated cutting-edge models available in Azure, specifically OpenAI’s GPT and Meta’s CodeLlama families.

Only the CodeLlama Instruct 70B model had comparable benchmarks to the GPT family, but it required a high-cost Azure machine with multiple GPUs. In contrast, Azure OpenAI provides a pay-as-you-go service, which, as of February 2024, included the latest GPT-4 Turbo⁶ model with a 128,000-token context window. This model performed better in benchmarks such as HumanEval [13] and MBPP [14] compared to CodeLlama Instruct 70B⁷. The ‘gpt-4-1106-preview’ version was offered at a promotional price, making it the preferred choice due to its performance and cost-effectiveness.

⁶<https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4>

⁷<https://huggingface.co/meta-llama/Llama-2-70b-chat-hf>

C. From User Stories to Acceptance Test Scenarios

This phase aims to derive acceptance tests from user stories, translating high-level requirements into detailed Gherkin scenarios to validate system behavior, ensure shared understanding among stakeholders, and guide implementation.

The inputs for this phase include:

- **User Story Title:** A concise, descriptive name capturing the feature or functionality from the user’s perspective.
- **User Story Description:** A detailed outline of functionality, including requirements, acceptance criteria, and expected outcomes.

The output is a set of acceptance test scenarios in Gherkin format, covering key aspects of the user stories for comprehensive validation.

To achieve accurate, comprehensive, and robust outputs across diverse scenarios, an iterative prompt engineering process was applied to optimize clarity, context awareness, and response specificity — see the final prompt in Appendix A.

D. From Acceptance Test Scenarios to Executable Test Scripts

The inputs for this phase are the user story to be tested, the corresponding Gherkin acceptance test scenarios, and the HTML code of the pages under test. The output is a Cypress test script automating all the specified Gherkin scenarios.

Initial experiments showed no need for fine-tuning, as the model generated valid TypeScript syntax for Cypress, likely due to the widespread adoption of this technology stack.

The following metrics were used to assess model’s performance and identify areas for improvement during the prompt engineering process:

- **Syntactic Correctness:** The proportion of generated tests adhering to TypeScript syntax.
- **Semantic Relevance:** The proportion of generated tests reflecting the inputted Gherkin scenarios.
- **Accessibility:** The proportion of generated tests with comprehensive natural language comments.
- **Prompt Size:** An evaluation of both input prompt and response sizes, for cost considerations.

The prompt engineering process involved the following steps: gathering a set of inputs, constructing a basic prompt, evaluating the performance metrics against the inputs, and iteratively modifying the prompt to enhance the metrics that exhibited poorer performance.

The final prompt consists of a user and a system prompt.

The user prompt contains information specific to the feature under test (to be collected automatically), including:

- **User Story:** For context on the feature being tested.
- **Gherkin Scenarios:** For context on the test cases to generate.
- **Pages’ HTML:** For context on the pages’ elements and their identifiers.

To reduce costs related to the prompt size, the HTML is preprocessed to remove style and script elements, as these were found unnecessary for the model’s understanding of the HTML, and did not compromise the performance metrics.

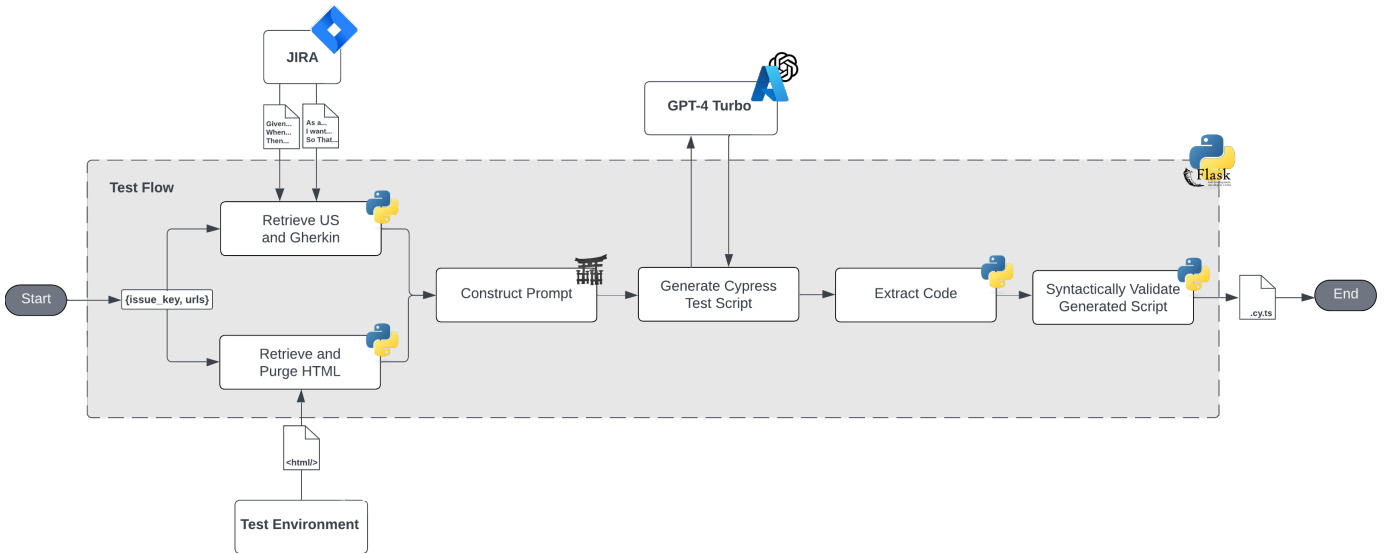


Fig. 2. Flow Diagram of Test Flow.

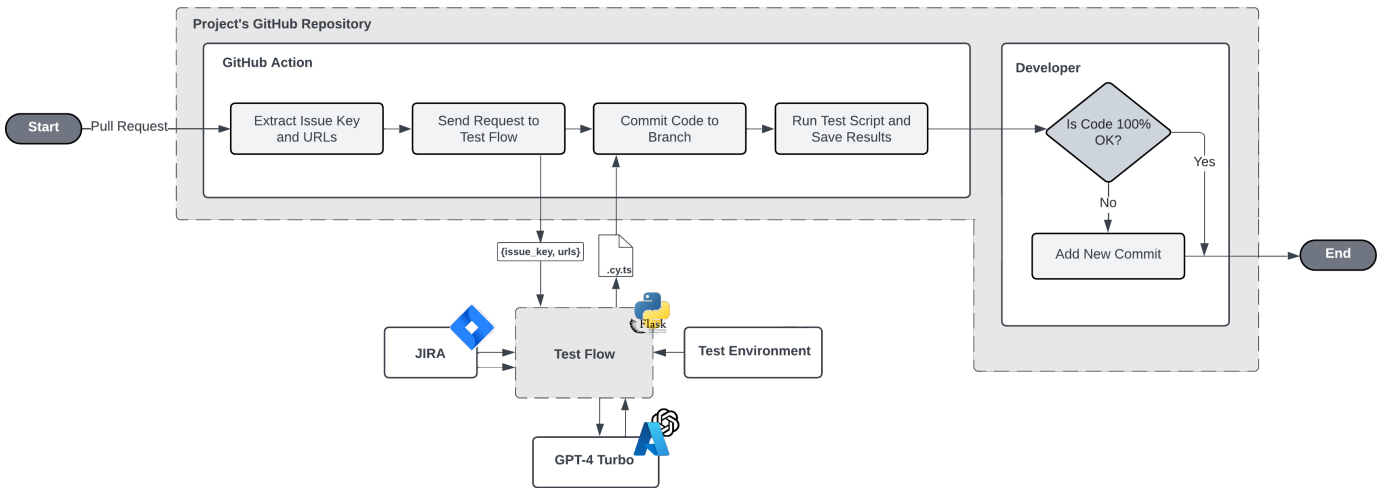


Fig. 3. GitHub Action for Integrating Test Flow into Current Workflows.

The system prompt provides additional context on the product and guides the model’s response, including (see Appendix B):

- **Product Context:** For additional context on the product being tested.
- **Cypress Good Practices:** For adherence to Cypress best practices.
- **Other Aspects:** For addressing specific situations the model was overlooking.

III. INTEGRATION IN THE INDUSTRIAL ENVIRONMENT

This section describes the tools developed to seamlessly incorporate our approach into our industrial partner’s operational framework, enabling its evaluation in a real-world context, whilst respecting confidentiality.

A. AutoUAT: From User Stories to Acceptance Test Scenarios

To integrate the process of transforming user stories into acceptance test scenarios in an industrial environment and gather user feedback and usage information, an automated workflow was developed using the Microsoft Power Platform.

The process begins with users filling out a form with the title and description of the user story. Its submission triggers an HTTP request to an Azure container where the LLM is hosted. The LLM processes the prompt provided by the tool with user story information and system instructions and generates the corresponding ATs. The generated ATs are then automatically sent back to the user via a private message on Microsoft Teams, who can also fill out a feedback form.

B. Test Flow: From Acceptance Test Scenarios to Test Scripts

Test Flow was developed for seamless integration into our partner’s workflow with the architecture depicted in Fig. 2.

To minimize user input and improve usability, Python modules were developed to automatically extract user stories and Gherkin test scenarios from JIRA⁸, given the issue key, and to retrieve HTML code of the pages under test stripped from irrelevant style and script elements, given their URLs. Then, the prompt is constructed following prompt engineering results and sent to GPT-4 Turbo. Upon receiving the model’s response, the generated code is extracted, validated for TypeScript syntax, and finally returned to the user.

Test Flow can be easily integrated with current workflows as it can be invoked through a Flask REST API. As an example of this integration, it was implemented a GitHub Action, illustrated in Fig. 3. Triggered on a new pull request (PR), it extracts the necessary inputs (issue key and URLs) from the PR description, sends a request with these inputs to Test Flow, commits the generated tests to the source branch of the PR, and executes them. As LLM outputs cannot be blindly trusted, developers must review the code generated by Test Flow and implement fixes, when necessary.

IV. EVALUATION

This section details the methods and results of the evaluation efforts conducted with our industrial partner, including experimental evaluations and user surveys. Additionally, it addresses the research questions posed at the beginning and discusses potential threats to the validity of our study. For confidentiality reasons, specific datasets used in this study are not disclosed.

A. AutoUAT Workshop

To gather initial user feedback about AutoUAT, we conducted a workshop with six experienced product owners who regularly write user stories and ATs. The session began with a 15-minute introduction to AutoUAT, followed by 15 minutes of hands-on use. Feedback was collected through a survey with 10 questions and an open comments section.

All participants had over three years of experience and were routinely involved in writing user stories (1 to 10 per sprint) and associated acceptance tests, ensuring feedback relevance.

Participants rated the quality of the AutoUAT outputs with an average high score of 8 out of 10 (Fig. 4). The likelihood of adoption was unanimously high, likely saving time and improving the acceptance criteria quality (Fig. 5).

A follow-up comment from a participant further highlights the tool’s impact: *“The tool is extremely helpful. I started writing user stories in Gherkin format, and this tool helped me identify missing acceptance criteria and refine design and UX. It saves me time, and I plan to continue using it.”*

B. AutoUAT Usage

In a second validation phase, the developed tool was made available to everyone within the company for a period of two months, and data was collected on its usage.

In total, the tool was utilized 166 times by the company’s professionals. After each use, users were prompted to provide feedback on the tool outputs. To the question “Was this

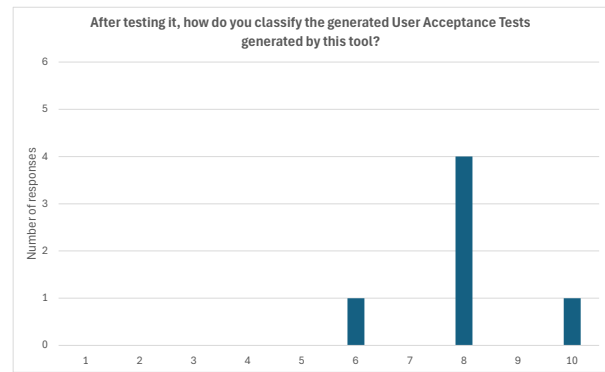


Fig. 4. Workshop Participants’ Feedback on AutoUAT Outputs.

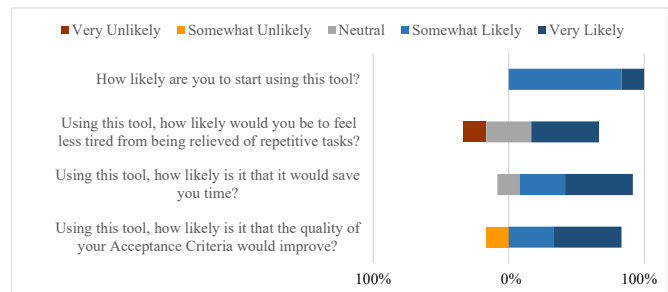


Fig. 5. Workshop Participants’ Feedback on AutoUAT Adoption and Benefits.

AT useful?”, only 3 respondents answered negatively, while 62 respondents answered positively, representing 95% of the respondents (Fig. 6).

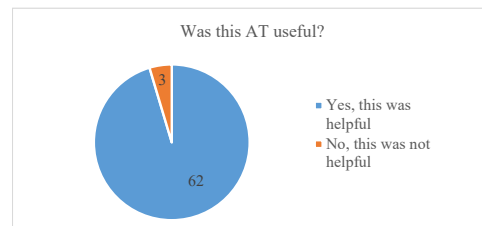


Fig. 6. Users’ Feedback on the AutoAUT Outputs.

Additionally, users were encouraged to leave additional comments to complement their responses. Some noteworthy comments include: *“In this situation, the tool allowed me to think about an error scenario that we were not considering. This had an impact on the product, allowing us to create an error screen in the UX.”*

In this case, the user requested the generation of acceptance test scenarios for the “Alphabet User Sign-Up” story described in Appendix C. Although the title lacks clarity, the model accurately captured the intended functionality (see Appendix D), demonstrating its flexibility in extracting functionality from context and tolerating minor errors. The last two scenarios demonstrate the model’s ability to extend beyond the explicit requirements of the user story.

⁸<https://www.atlassian.com/software/jira>

C. Test Flow Experimental Evaluation

The validation process for Test Flow started with a quantitative evaluation of the tool for the performance metrics defined in Section II-D, using selected issues from a company’s e-commerce product.

In order to ensure data quality and relevance, JIRA issues were selected based on the following criteria:

- **Story Classification:** The issue must be classified as a Story.
- **User Perspective:** The story must be written from the user’s perspective.
- **Gherkin Tests:** The issue must have acceptance tests in Gherkin (handwritten or generated with AutoUAT).
- **Quality Assessment:** The issue must have medium to high quality.⁹
- **Team Assignment:** The issue must be preferably assigned to the teams we were most in contact with.¹⁰

Given the product’s maturity and its current focus on UI modifications and defect resolution rather than new feature development, the sample size was inevitably limited. Nevertheless, we managed to compile a final dataset of 13 issues, each with their US description and Gherkin test scenarios, and gathered the corresponding page URLs for each of them.

In order to validate the product flow comprehensively, we selected issues that targeted a diverse set of pages. Table I illustrates the distribution of the selected issues and Gherkin scenarios according to the related page.

TABLE I
INPUT SUBJECTS FOR THE TEST FLOW EXPERIMENTAL EVALUATION.

Web Page	#Stories	#Gherkin Scenarios
Product List Page	2	9
Product Detail Page	3	10
Cart Page	2	7
Checkout Page	4	16
Orders Page	2	8
Total	13	50

As output, Test Flow generated a Cypress test script for each issue (user story), resulting in 13 scripts and 50 test cases (one per Gherkin scenario) in total. The 50 test cases generated were rigorously evaluated, through a detailed manual analysis, against the previously defined performance metrics. Concurrently, strategies were devised and implemented to rectify the erroneously generated test cases, providing insights into the effort required to correct these test cases. The workflow for this framework and respective results is depicted in Fig. 7. The results for the performance metrics are as follows:

- **Syntactic Correctness:** 100% of the generated test cases adhered to TypeScript syntax.
- **Semantic Relevance:** 60% of the generated test cases accurately mirrored the original Gherkin scenarios.

⁹This was subjectively determined based on whether the US provided sufficient context, was unambiguous, and maintained consistent naming conventions.

¹⁰To facilitate the resolution of potential queries and the validation of outputs.

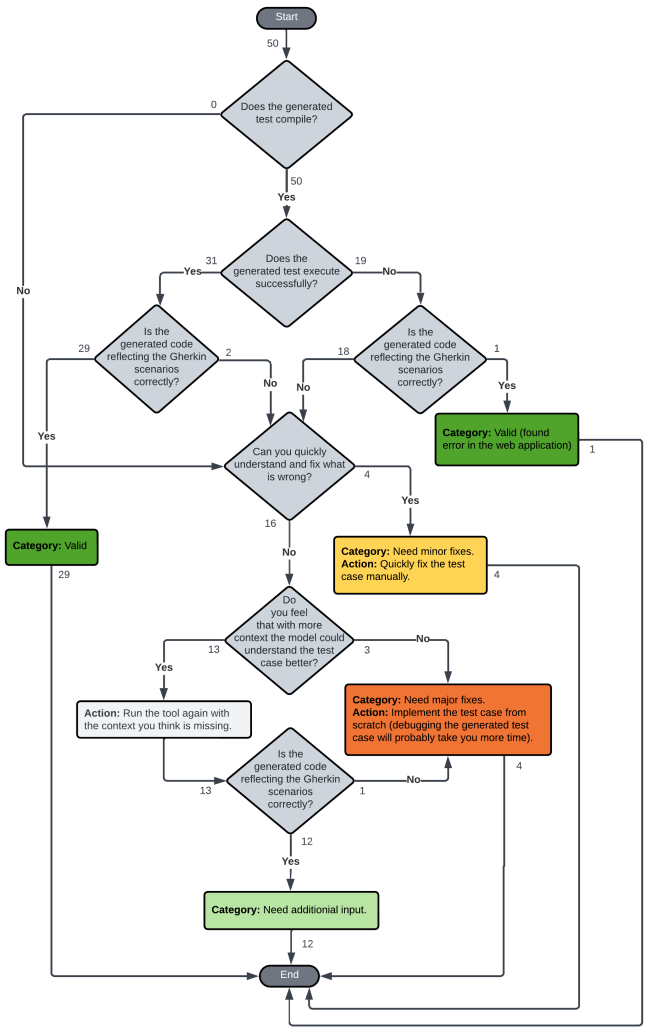


Fig. 7. Procedure for evaluating and classifying the test cases generated with Test Flow. The arrows are labelled with the number of test cases that adhered to the corresponding path in our experiment.

- **Accessibility:** 100% of the generated test cases had comprehensive NL comments.
- **Prompt Size:** Input prompts averaged 9500 tokens; output prompts averaged 750 tokens.

The tests generated were syntactically correct and included detailed NL comments, demonstrating the solutions’s robustness and accessibility. However, the evaluation revealed improvement areas, particularly in semantic relevance, as 40% of test cases did not accurately reflect the Gherkin scenarios. Manual analysis categorized these test cases into three types:

- **Lack of Context** (12 cases): Inputted US and Gherkin scenarios lack crucial context for the model to correctly understand the functionality to test.
- **Minor Errors** (4 cases): Errors of less than one line of code that require less than two minutes per test case to comprehend and rectify.
- **Complex Errors:** (4 cases) Remaining errors. In these cases, it is considered not worth the effort to fix them.

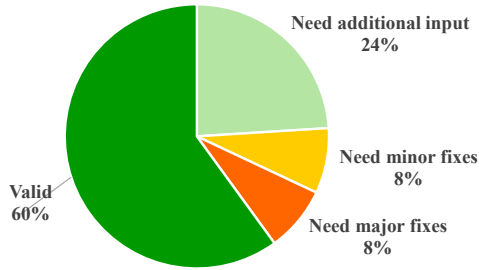


Fig. 8. Classification of 50 Executable Test Cases Generated by Test Flow.

Fig. 8 provides a visual representation of this distribution. The tests within the first two categories constitute 32% of the total generated test cases. As depicted in Fig. 7, these can be rectified by enriching the input context and implementing minor fixes (one line of code), respectively. Implementing these strategies allowed us to elevate the semantic relevance to 92%, leaving only the tests with complex errors. In such cases, manual implementation from scratch is recommended.

Furthermore, a particular failed test revealed a discrepancy; the US and Gherkin tests suggested that a button should be disabled, whereas it was actually hidden. This discrepancy pointed to a flaw in the application being tested, demonstrating Test Flow’s defect detection capability.

The size of input and output prompts influences the cost of using Test Flow. Based on the pricing of Azure OpenAI’s GPT-4 Turbo as of June 2024, the average cost for generating a test script per user story was 0.12€ in this experiment.

Our investigation found no correlation between specific pages and poorer test outcomes. Instead, test quality was more closely linked to the clarity and context provided in the user stories and Gherkin scenarios.

D. Test Flow Workshop and User Survey

Following the experimental evaluation of Test Flow, a workshop was conducted with the company’s developers. It included a 15-minute presentation on Test Flow’s capabilities, a 15-minute demo of inputs and outputs, and a brief 3-minute survey, which received 16 responses.

The survey targeted individuals who typically conduct Cypress testing, including developers (responsible for creating tests for the features they develop) and Scrum Masters (who also function as developers). Respondents could optionally disclose their identity for follow-up; 11 did so.

Participant demographics are shown in Table II, with most respondents being developers of an e-commerce product team, with a wide range of experience levels.

To evaluate Test Flow, ten statements were presented on a Likert scale from “Strongly Disagree” to “Strongly Agree.” Respondents rated the solution’s intuitiveness, accessibility, workflow integration, and potential to accelerate Cypress test script creation. Fig. 9 shows the survey questions and results.

The statements were designed to ensure response validity by including both positively and negatively phrased questions.

TABLE II
DEMOGRAPHICS OF 16 TEST FLOW SURVEY RESPONDENTS.

Current Role	Experience	Curr. Prod.
Developer	11	E-commerce 15
ScrumMaster	4	Other 1
Intern	1	
	< 1 year	2
	1-3 years	8
	4-5 years	4
	> 5 years	2

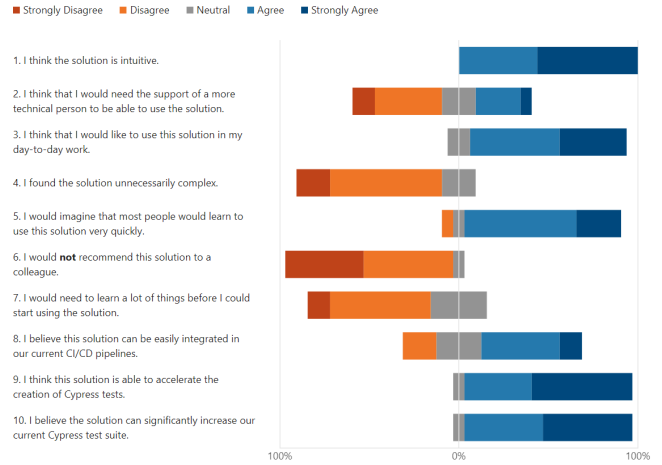


Fig. 9. Results of the Test Flow User Survey (16 Respondents).

The overall feedback was highly positive, with consensus on 8 out of the 10 statements. Respondents found the solution intuitive, accessible, and believed it could be easily integrated into their workflows, enhancing the creation of Cypress tests.

However, statements 2 and 8 received mixed responses, leading to follow-up discussions with 11 respondents who opted for further engagement:

- For statement 2, “I think that I would need the support of a more technical person to be able to use the solution”, four respondents initially agreed, and one strongly agreed. Upon follow-up, the four clarified that they misunderstood the question and found the solution user-friendly. The strongly agreeing respondent was anonymous and could not be contacted.
- For statement 8, “I believe this solution can be easily integrated into our current CI/CD pipelines”, three respondents disagreed, with only one available for follow-up. Feedback indicated the issue stemmed from specific CI/CD pipeline characteristics rather than the solution itself, suggesting that integration may require customization based on project needs.

During follow-up, participants also expressed concerns about data security when using Generative AI, as the use of company-hosted models was not emphasized in the presentation. Clarifying this point reassured participants.

Overall, Test Flow’s validation provided valuable insights, highlighting its strengths and areas for improvement, including enhancing adaptability, refining integration support, and clearly communicating security measures.

E. Answer to Research Questions

Following the evaluation of both tools, we can address the research questions outlined in Section I.

RQ1: To what extent can LLMs help in generating accurate and comprehensive acceptance test scenarios in natural language from user stories?

In 65 usage instances, the users considered the ATs generated by AutoUAT helpful 95% of the time. AutoUAT also received very positive feedback from 6 product owners.

We conclude that **AI-powered assistants like AutoUAT have a strong potential to help product owners and other stakeholders in creating high-quality acceptance test scenarios from user stories with less effort.**

RQ2: To what extent can LLMs help in generating accurate and complete executable test scripts from acceptance test scenarios in natural language?

Test Flow generated accurate and complete executable test scripts from acceptance test scenarios in NL (with one test case per test scenario) 60% of the time, improved to 92% with minor fixes (8%) or additional contextual inputs (24%). The generated scripts contained detailed comments in natural language to facilitate their (mandatory) review by developers responsible for test automation. Survey feedback was unanimously positive, emphasizing Test Flow’s potential to speed up Cypress test creation and integrate smoothly into workflows.

We conclude that **AI-powered assistants like Test Flow have a strong potential to help developers in the translation of NL acceptance test scenarios to executable test scripts over the application UI with less effort.**

However, further studies in industrial settings are needed to better assess the long-term productivity and quality gains.

F. Threats to Validity

Several factors may limit the generalizability of our findings:

- **Context-Specific Results:** Evaluation was performed within a single company with strong quality processes. The approach may yield different outcomes in other environments or domains, especially where process or product standards are lacking (e.g., *data-test-ids* for testability).
- **Sample Size Limitations:** The relatively small number of user stories tested with Test Flow and AutoUAT may affect the generalizability of results. A larger, more varied sample across different projects would enhance reliability.
- **Short Evaluation Period:** The two-month evaluation may miss long-term usability issues, evolving user behaviors, and learning effects.
- **Feedback Bias:** Out of 166 usage instances of AutoUAT, feedback was provided for only 65 instances, which may not represent all users. Positive experiences might be overrepresented, skewing acceptance rates.

These limitations indicate a need for further research and validation across diverse industrial settings to fully understand the approach’s effectiveness and constraints.

V. RELATED WORK

This section reviews existing literature and approaches related to our work, highlighting their results, limitations, and the unresolved challenges they present. By analysing these approaches, we aim to underscore the gaps our methodology addresses.

A. From User Stories to NL Acceptance Test Scenarios

The literature addressing automatic AT generation, specifically using LLMs within a BDD framework, is limited. In fact, in [6], the authors identify several studies applying LLMs to unit and system testing but note that “there is still no research on the use of LLMs in integration testing and acceptance testing”, underscoring a research gap our work seeks to fill.

A notable example is a recent study [15] that evaluates the capabilities of several LLMs (GPT-3.5, GPT-4, Llama-2-13B, and PaLM- 2) for generating syntactically correct Gherkin scenarios from user stories. In an experiment with 50 user stories extracted from various sources, both GPT models demonstrated superior performance, with syntax errors in only one out of 50 generated feature files. They also manually checked some generated acceptance tests for semantic validity.

In comparison, our approach uses the more recent GPT-4 Turbo model and includes a more comprehensive assessment of the generated Gherkin scenarios by tool users (typically product owners) in a real-world industrial setting, as well as initial feedback on potential performance benefits, thus providing empirical insights into the practical impact of LLMs for BDD. This industry-focused evaluation highlights the novelty and relevance of our approach.

B. From Acceptance Test Scenarios to Executable Test Scripts

We identified two recent studies [16], [17] that examine productivity gains from using AI assistants and other acceleration technologies for generating test scripts compared to manual methods. Both studies start with test cases specified in Gherkin, which are then converted into executable tests.

In [16], the authors compare the effort to create Web end-to-end (E2E) test scripts based on natural language test descriptions using several approaches:

- Manual - test scripts are crafted manually by the tester;
- ChatGPT Lite - the tester prompts ChatGPT to generate a test script based on the test description, copies the generated script to the IDE and manually refines it as needed, without further interaction with ChatGPT;
- ChatGPT Max - similar to Lite, but the tester may engage in a conversation with ChatGPT to refine the initial test script, before copying it to the IDE;
- Copilot - inside the IDE, the tester starts by providing the test description in natural language, which guides Github Copilot in the incremental generation of code suggestions, which are reviewed and edited as needed.

They used Gherkin for the description of the test scenarios in natural language, Java as the scripting language, JUnit as the test framework, and Selenium WebDriver as the browser automation library.

In an experiment involving 1 junior tester, 8 real-world Web applications, and 12 Gherkin test cases per application (crafted by one of the authors), they found that the Manual approach took the longest (median of 54 seconds per Gherkin line of code), while ChatGPT Max required the least time (38 seconds per line), with statistically significant productivity gains. ChatGPT Lite and Copilot had quite similar results (medians of 47 and 45 seconds per line, respectively).

In contrast, Test Flow’s prompt includes the HTML code of the pages under test, drastically reducing the need for manual refinements to provide correct locators of UI elements, leading to potentially higher productivity gains.

In [17], the authors compared the manual effort required for three Web test automation approaches:

- Programmable (PT) - manually implemented test cases;
- Capture & Replay (CRT) - captures manual interactions with a tool for later replay;
- NLP-based (NLT) - describes test steps in a domain-specific language or restricted natural language, interpreted by a test automation tool, with special constructs for identifying the UI elements.

The PT approach used Selenium WebDriver as the browser automation tool, Java as the programming language and JUnit as the test framework; the CRT approach used Selenium IDE to capture the user interactions, and NLT used an unspecified tool.

They conducted an experiment with 9 real-world Web applications with multiple releases, 3 junior testers/developers, and a set of Gherkin test cases written by one of the authors for each application (used as input specification for all approaches).

For the initial test suite development, PT required the most effort, CRT the least, with NLT in between. Over multiple releases, NLT had the lowest cumulative effort, except in one high-complexity case. These results reflect CRT’s low script reusability and NLT’s limited flexibility for high-complexity applications. Thus, the authors conclude that NLT is well-suited for low to medium-complexity applications.

In contrast, Test Flow generates test scripts automatically from Gherkin test specifications with minimal user intervention. This is expected to reduce the initial test development effort significantly. In cases where automatic generation fails, users can resort to other approaches.

C. Other Approaches

We found a work [18] that tries to use LLMs and ML to test Web applications through their GUI in a fully automatic way. However, the approach is closer to a monkey testing approach and not a specification-based one. They use GPT-4 and Selenium WebDriver for automatically exploring and testing Web applications in an interactive way without user intervention. In each step, they use Selenium WebDriver to capture the current state of the GUI and identify possible next actions on the GUI. Then, they prompt GPT-4 to select the next GUI action based on the following information: a generic test goal (such as maximizing the number of actions and states

explored), the HTML code of the Web application, filtering out irrelevant formatting and behavioural information; the history of past actions in previous iterations; the possible next actions. The selected action is then performed in the AUT via Selenium WebDriver. This process is repeated until a specified number of interaction steps is reached.

In contrast, our approach is specification- and script-based, allowing for more comprehensive and repeatable specification coverage and error detection. Test Flow generates test scripts based on requirement specifications (user stories) and test specifications (Gherkin scenarios, including expected outputs), which, in turn, may be semi-automatically generated using AutoUAT. Overall, both approaches are complementary.

A few studies [19], [20] have explored generating executable tests from Gherkin test scenarios using NLP and automated reasoning techniques, but their targets were Java classes or Python functions: [19] achieved a 73% success rate on a sample of 12 scenarios from tutorials, while [20] successfully translated 80% of scenarios from 20 open-source projects used to develop the approach and 17% from an additional 50 open-source projects. In contrast, our approach focuses on generating tests for Web applications through the GUI, posing distinct challenges, such as identifying target UI elements.

In [21], the authors introduce XUAT-Copilot, an LLM-powered tool for automating AT of mobile applications, specifically WeChat Pay. The tool takes a set of test cases composed of parameterized steps in natural language (similar to Gherkin) and corresponding test data as input. Three LLM-based agents—responsible for action planning, state checking, and parameter selection—collaborate to interpret these steps and translate them into UI actions on the app via an intermediate “skill” library, generating corresponding test scripts. In an experiment with a sample of passing test cases, XUAT-Copilot achieved 88.55% success in test case translation (into passing test scripts), and 93.03% in test step translation.

In contrast, our approach targets Web applications, generating test scripts directly knowing the HTML of the pages under test, for a lighter solution. Although our approach already handles scenarios that involve navigation between multiple pages distinguishable by their URLs, web pages that dynamically change their structure and single-page applications may benefit from a more complex approach like XUAT-Copilot.

VI. CONCLUSIONS AND FUTURE WORK

This section summarizes our key findings and suggests future research directions to enhance the use of LLMs in acceptance testing.

A. Conclusions

Our study demonstrates the potential of LLMs to improve the acceptance testing process, by automatically generating acceptance test scenarios and executable scripts from user stories. The solution, leveraging GPT-4 Turbo, was designed to streamline Gherkin scenario creation from user stories and Cypress script generation from these scenarios. Both tools received positive feedback, with AutoUAT rated highly

by experienced product owners and Test Flow noted for its usability and effectiveness in automating test creation.

AutoUAT achieved a 95% acceptance rate in AT generation, covering both explicit requirements and additional scenarios. Test Flow generated test scripts with 60% initial accuracy, rising to 92% with minor refinements, thus reducing developer workload in creating and verifying test scripts.

While effective, integrating LLMs for AT generation has challenges, such as ensuring detailed contextual inputs (namely, product and UI descriptions), managing potential errors, and addressing cost concerns. Despite these challenges, our results suggest that LLMs can significantly improve the acceptance testing process, with time and cost savings.

B. Future Work

This study suggests several directions for future research in the context of AI-assisted acceptance testing: training models on domain-specific datasets to enhance AT accuracy, integrating AT generation with project/issue management tools, exploring the impact of detailed user stories on test quality, optimizing costs by reducing input token requirements, conducting broader studies across projects and industries, and conducting long-term studies to better assess productivity and quality benefits.

APPENDIX

A. Structure of AutoUAT's prompt

system:

You are an expert assistant specializing in creating User Acceptance Tests using Gherkin language. Your behavior should be that of a meticulous and detail-oriented professional, dedicated to producing clear, comprehensive, and precise User Acceptance Tests. Your objective is to generate high-quality User Acceptance Tests based on the provided User Story titles and descriptions, ensuring no ambiguities. The tests should be thorough and follow the Gherkin syntax accurately.

Context:

The User Acceptance Tests you generate will be used by a development team to validate that the functionality of the application meets the specified requirements. It is crucial that the tests cover various scenarios, including edge cases, to ensure robust validation. The User Stories provided will include a title and a description, and your task is to translate these into Gherkin language tests.

```
{% for item in chat_history %}
user:
Here is a User Story title and description.
Generate User Acceptance Tests in Gherkin
language for this User Story.
```

```
User Story Title: {{item.inputs.title}}
User Story Description: {{item.inputs.
description}}
```

```
Here are the User Acceptance Tests in Gherkin
language for the given User Story:
assistant:
{{item.outputs.answer}}
{% endfor %}
```

B. Structure of Test Flow's system prompt

(...Product context information omitted for confidentiality reasons...)

You will receive a user story, Gherkin scenarios and HTML of the pages to test. You are responsible for writing the Cypress tests for the Gherkin scenarios.

Keep in mind the following best practices:

- You generate the test to be as complete as possible for the scenario.
- You use the data-test-id to locate the element if you need to interact with it.
- Keep tests independent, so they can run in any order.
- Use Cypress built-in assertions.

You already have some Cypress commands and variables to use in your tests:

```
```typescript
(...omitted...)
```
```

Some notes:

- You output the Cypress code only and inside a markdown code block.
- If any additional information is needed, put it in a comment inside the code block.
- Pay attention to the html provided when writing the tests so that you can use the correct data-testid and Cypress commands to interact with the elements.
- When using text to assert an element's content, pay attention to the language of the page (...).
- Ensure your code includes comments that guide through the steps for code understanding and accessibility.

C. Example of User Story

Title: Alphabet User Sign-Up

Description:

- **As an** Alphabet App user,
I want to access my profile,
So that I can allow for the collection of Analytics.

When the user is in the Profile & Settings Main Page, in the Legal Information section the user can activate or deactivate a toggle to allow the collection of data (analytics). This will communicate with the Backend to guarantee the activation for analytics tracking.

Actions & Error Handling:

- The user clicks on the toggle. It moves to the right and becomes green to activate.

- The user clicks on the toggle. It moves to the left and becomes grey to deactivate.

D. Example of Generated Acceptance Test Scenarios

Feature: Legal Information - Usage Data Tracking

Scenario: User activates the collection of analytics data

- **Given** the user is on the Profile & Settings Main Page
And the user is in the Legal Information section
When the user clicks on the data collection toggle
Then the toggle should move to the right
And the toggle should turn green
And the backend should be notified to activate analytics tracking

Scenario: User deactivates the collection of analytics data

- **Given** the user is on the Profile & Settings Main Page
And the user is in the Legal Information section
When the user clicks on the data collection toggle
Then the toggle should move to the left
And the toggle should turn grey
And the backend should be notified to deactivate analytics tracking

Scenario: User tries to activate the collection of analytics data but the backend fails

- **Given** the user is on the Profile & Settings Main Page
And the user is in the Legal Information section
When the user clicks on the data collection toggle
And the backend fails to activate analytics tracking
Then the toggle should not move to the right
And the toggle should not turn green
And an error message should be displayed

Scenario: User tries to deactivate the collection of analytics data but the backend fails

- **Given** the user is on the Profile & Settings Main Page
And the user is in the Legal Information section
When the user clicks on the data collection toggle
And the backend fails to deactivate analytics tracking
Then the toggle should not move to the left
And the toggle should not turn grey
And an error message should be displayed

REFERENCES

- [1] T. B. Brown, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.
- [2] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot," 2023, available at <https://arxiv.org/abs/2302.00438>.
- [3] M. Wermelinger, "Using GitHub Copilot to solve simple programming problems," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 172–178.
- [4] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of ai on developer productivity: Evidence from github copilot," *arXiv preprint arXiv:2302.06590*, 2023.
- [5] BlueOptima Limited, "The impact of generative ai on software developer performance," BlueOptima Limited, Tech. Rep., 2024, © BlueOptima Limited 2005–2024. All Rights Reserved. [Online]. Available: <https://www.blueoptima.com/resource/dora-lead-time-to-change-useful-but-inadequate/>
- [6] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, 2024.
- [7] M. Bolton, "User acceptance testing – a context-driven perspective," in *Twenty-fifth Annual Pacific Northwest Software Quality Conference. The Conference*, 2007, pp. 535–548.
- [8] M. Wynne and A. Hellesøy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. The Pragmatic Bookshelf, 2017.
- [9] M. Gärtner, *ATDD by example: a practical guide to acceptance test-driven development*. Addison-Wesley Professional, 2013.
- [10] M. Moe, "Comparative study of test-driven development tdd, behavior-driven development bdd and acceptance test-driven development atdd," *International Journal of Trend in Scientific Research and Development*, pp. 231–234, 06 2019.
- [11] B. García, J. M. del Alamo, M. Leotta, and F. Ricca, "Exploring browser automation: A comparative study of selenium, cypress, puppeteer, and playwright," in *International Conference on the Quality of Information and Communications Technology*. Springer, 2024, pp. 142–149.
- [12] B. Haugset and G. K. Hanssen, "Automated acceptance testing: A literature review and an industrial case stud," in *Agile 2008 Conference*. IEEE, 2008, pp. 27–38.
- [13] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.
- [14] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.
- [15] S. Karpurapu, S. Myneni, U. Nettur, L. S. Gajja, D. Burke, T. Stiehm, and J. Payne, "Comprehensive evaluation and insights into the use of large language models in the automation of behavior-driven development acceptance test formulation," *IEEE Access*, vol. 12, pp. 58 715–58 721, 2024.
- [16] M. Leotta, H. Z. Yousaf, F. Ricca, and B. Garcia, "AI-Generated Test Scripts for Web E2E Testing with ChatGPT and Copilot: A Preliminary Study," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 339–344.
- [17] M. Leotta, F. Ricca, A. Marchetto, and D. Olianias, "An empirical study to compare three web test automation approaches: NLP-based, programmable, and capture&replay," *Journal of Software: Evolution and Process*, vol. 36, no. 5, p. e2606, 2024.
- [18] D. Zimmermann and A. Koziolok, "GUI-Based Software Testing: An Automated Approach Using GPT-4 and Selenium WebDriver," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 2023, pp. 171–174.
- [19] S. Kamalakar, S. H. Edwards, and T. M. Dao, "Automatically generating tests from natural language descriptions of software behavior," in *International Conference on Evaluation of Novel Software Approaches to Software Engineering*, vol. 2. SCITEPRESS, 2013, pp. 238–245.
- [20] T. Storer and R. Bob, "Behave nicely! automatic generation of code for behaviour driven development test suites," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2019, pp. 228–237.
- [21] Z. Wang, W. Wang, Z. Li, L. Wang, C. Yi, X. Xu, L. Cao, H. Su, S. Chen, and J. Zhou, "Xuat-copilot: Multi-agent collaborative system for automated user acceptance testing with large language model," *arXiv preprint arXiv:2401.02705*, 2024.