

UNIVERSITY OF SCIENCE, VIETNAM NATIONAL UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

Nguyễn Trịnh Như Ý - Phạm Ngọc Anh Thư

**GENERATING MANUAL TEST CASES
FROM USE CASES USING LARGE
LANGUAGE MODELS**

BACHELOR OF SCIENCE IN INFORMATION TECHNOLOGY
HIGH-QUALITY PROGRAM

Hồ Chí Minh City, 07/2024

UNIVERSITY OF SCIENCE, VIETNAM NATIONAL UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

Nguyễn Trịnh Như Ý - 20127100

Phạm Ngọc Anh Thư - 20127343

**GENERATING MANUAL TEST CASES
FROM USE CASES USING LARGE
LANGUAGE MODELS**

BACHELOR OF SCIENCE IN INFORMATION TECHNOLOGY
HIGH-QUALITY PROGRAM

SUPERVISOR

Assoc. Professor Nguyễn Văn Vũ

Hồ Chí Minh City, 07/2024

Acknowledgment

First of all, we would like to express our gratitude to the University of Science, Vietnam National University, Ho Chi Minh City, for providing the time and resources for this thesis.

Furthermore, we sincerely thank all the professors at the University of Science, Vietnam National University, Ho Chi Minh City, for equipping us with valuable knowledge over the past four years, enabling us to complete this thesis. We cannot be more grateful for all the lessons imparted and the contributions made by the professors.

With all the dedication and effort over the past period, along with the knowledge we have gained at the university, we have completed this graduation thesis. However, due to limited knowledge and practical experience, we cannot avoid mistakes. Therefore, we sincerely hope for the understanding and feedback from the esteemed professors and fellow students.

Especially, we would like to express our deepest gratitude to Associate Professor Nguyễn Văn Vũ, our advisor, for sharing his knowledge, assisting, encouraging, and inspiring us.

We sincerely thank you.

UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY
COMMENTS FROM THESIS'S ADVISOR
(Research)

Thesis's title: GENERATING MANUAL TEST CASES FROM USE CASES USING LARGE LANGUAGE MODELS

Students: **Nguyễn Trịnh Như Ý** (20127100) – **Phạm Ngọc Anh Thư** (20127343)

Advisor: Associate Professor Nguyen Van Vu, Ph.D.

1. Research Topic and Idea

In this thesis project, the team investigated an approach for generating test cases for manual testing (in natural language) from use-case specifications using large language models (LLMs). This topic is significant for several reasons. First, a considerable amount of software testing today is still performed manually. Second, testers typically create test cases based on requirements before executing them on the application under test, and this test-case design process can be both time-consuming and prone to errors. Automating this process can enhance software quality and reduce the time spent on testing activities. Moreover, many software projects document requirements using use-case specifications. Finally, recent advancements in LLMs provide opportunities to automate the generation of technical documents for software development, including test cases from use-case specifications.

2. Research Methodology

The proposed method for generating test cases consists of two phases, test design and test case generation. The test design phase focuses on generating sets of test scenarios to best reflect conditions and situations covered in use cases. The second phase focuses on producing detailed test steps for each test scenario. Test scenarios and test steps are written in natural language so that testers can read and follow them. The team applied GPT-4's prompting techniques to produce test cases.

To evaluate the proposed method, the team applied it to a dataset of four real-world applications, consisting of 44 use cases. The outputs were assessed using metrics such as correctness, duplication, incorrectness, and coverage.

Overall, the research method was carried out well.

3. Contributions

The study offers several contributions. One is a new method for generating test cases for manual testing from use-case specifications. The generated use cases are reasonably accurate and readable by testers. Another contribution is an experiment to provide evidence that it is feasible to use LLMs for automated generation of test cases for manual testing.

4. Management and Progress Reporting

The students have been highly proactive throughout the project. They actively listened to feedback and conducted thorough investigations based on the guidance provided. As diligent individuals, they did not hesitate to undertake the tasks I recommended. I am exceedingly pleased with their performance.

5. Thesis Writing

The thesis is well-organized and clearly written. It provides sufficient detail on the proposed method and the experimental design used to evaluate it. The results are thoroughly analyzed, interpreted, and discussed.

6. Oral Presentation

The students delivered an excellent presentation in front of the committee, demonstrating both confidence and a clear understanding of their accomplishments throughout the thesis. They responded to the committee's questions comprehensively.

7. Publication and/or Real-world Application

A paper was prepared and submitted to the Pacific Rim International Conference on Artificial Intelligence (PRICAI 2024). However, by the time the students presented their thesis, the result of the submission had not yet been announced

Rank: *Outstanding*

Ho Chi Minh city, August 16th, 2024

Advisor



Nguyen Van Vu

BẢN NHẬN XÉT KHÓA LUẬN TỐT NGHIỆP

(HƯỚNG NGHIÊN CỨU)

Tên đề tài: Generating test cases from use cases using large language models

Sinh viên thực hiện: 20127100 – Nguyễn Trịnh Như Ý

20127343 – Phạm Ngọc Anh Thư

Giảng viên phản biện: TS. Trần Duy Hoàng

1. Chủ đề và ý tưởng nghiên cứu:

Việc tạo ra các test cases từ đặc tả yêu cầu là một công việc thủ công, gây mất nhiều thời gian của tester. Nghiên cứu hướng tới việc phát sinh các test case một cách tự động từ các đặc tả use case bằng ngôn ngữ tự nhiên. Nghiên cứu đề xuất áp dụng kỹ thuật prompt engineering trên các mô hình ngôn ngữ lớn được đào tạo trước (pre-trained large language models), cụ thể là GPT-4 và BERT.

2. Phương pháp nghiên cứu:

Nhóm đã khảo sát, nghiên cứu các công trình liên quan đến bài toán phát sinh test case từ đặc tả use case. Đề xuất mô hình gồm 2 pha chính: phát sinh test scenario từ use case và phát sinh test case từ test scenario; trong đó mỗi pha chính lại bao gồm nhiều bước tiền xử lý, phát sinh từng thành phần và hậu kiểm sử dụng kỹ thuật prompt engineering trên GPT-4 và BERT. Nhóm đã thu thập các đặc tả use case của các ứng dụng trong thực tế. Thực hiện thực nghiệm để đánh giá chất lượng cũng như độ phù hợp của test case được sinh ra.

3. Đóng góp Khoa học và thực tiễn:

Đóng góp chính của nghiên cứu là hướng tiếp cận chia để trị. Trong đó, thay vì sử dụng một câu prompt để phát sinh đối tượng từ dữ liệu đầu vào, quá trình được thực hiện qua nhiều bước bao gồm tiền xử lý giúp phân rã nội dung đầu vào, phát sinh từng thành phần của test scenario và hậu xử lý để đánh giá chất lượng của đối tượng được phát sinh. Thực nghiệm chứng minh mô hình nhiều bước cho kết quả tốt hơn so với mô hình một bước phát sinh đối tượng trực tiếp từ thông tin đầu vào. Tuy nhiên, nghiên cứu còn một số hạn chế: mô hình GPT-4 được lựa chọn mà chưa có sự so sánh với các mô hình ngôn ngữ lớn khác; các đánh giá mang tính chủ quan vì không có dữ liệu nền tảng (ground truth) để đối sánh.

Giảng viên phản biện - 1

4. Báo cáo viết:

Báo cáo viết được trình bày bằng tiếng Anh, bao gồm 5 chương được trình bày rõ ràng, logic và có hệ thống. Các phần của báo cáo từ giới thiệu bài toán, nền tảng lý thuyết, phương pháp đề xuất đều được trình bày một cách chi tiết và dễ hiểu. Tuy nhiên, trong chương thực nghiệm nên làm rõ định nghĩa của các độ đo dùng để đánh giá, tránh gây nhầm lẫn cho người đọc.

5. Trình bày trước hội đồng:

Nhóm đã trình bày một cách rõ ràng về bài toán, hướng tiếp cận cũng như kết quả thực nghiệm của nghiên cứu.

6. Công bố khoa học/ ứng dụng thực tế:

Nhóm có nộp bài cho hội nghị PRICAI 2024 và đang chờ kết quả.

Đánh giá xếp loại: Giỏi

TP.HCM, ngày 15 tháng 8 năm 2024

Giảng viên phản biện

(Ký và ghi rõ họ tên)



Trần Duy Hoàng

Giảng viên phản biện - 2



fit@hcmus

**UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION
TECHNOLOGY**

THESIS PROPOSAL

GENERATING MANUAL TEST CASES FROM USE CASE SPECIFICATION USING LARGE LANGUAGE MODEL

1. GENERAL INFORMATION

Supervisor

– Assoc. Prof. Nguyễn Văn Vũ (Faculty of Information Technology)

Students group:

1. Nguyễn Trịnh Như Ý (Student's ID: 20127100)

2. Phạm Ngọc Anh Thư (Student's ID: 20127343)

Type of topic: Research

Research time: From January 2024 to July 2024

2. RESEARCH CONTENT

2.1 Introduction

In the current time, Large Language Models (LLMs) are increasingly developing and can support humans in various tasks. GPT stands for Generative Pre-trained Transformer, is a well-known model that has been applied to a variety of tasks that require natural language processing. This includes automating the software development process to cut the loss of effort and time at this stage.

For the software development process, generating a manual test case set in the initial phase is extremely difficult when ensuring coverage and performance even without an application. In this situation, the work of writing manual test cases and testing states, which is mostly on the description of use cases and commonly done by humans. This process takes plenty of time and effort.

With the appearance of Large Language Model such as GPT-4, this research suggests leveraging the GPT-4 technologies to automate the process of generating manual test cases set based on use case specification to address the above issues.

2.2 Research objectives

Automating the process of creating manual test suites helps ease the workload of testers. In this phase, when it is not possible to create test cases based on real applications, reading the detailed use case specification is important. But this task causes difficulties and makes it easy to make mistakes. The development of Large Language Models, such as GPT, gives the opportunity to automate this process and cut the burden for testers.

This topic aims to produce a manual test case set in the initial stage of the software development process. Thanks to the test case set, enterprises can predict the amount of time, effort, and cost of the testing in the early stages.

About the contributions, this topic can enhance the software testing process, reduce the dependence on human resource and boost the efficiency of the software development.

2.3 Scope

The main concern of this topic is the application of GPT-4 to produce a set of test cases written in natural language.

Research subjects involves:

- GPT-4 technology: This research applies GPT-4 to automate the process of generating manual test cases from use case specifications.
- Use case specification: Defining the necessary information in use case specification that need to be extracted and provided to GPT-4 to generate the most efficient test cases.
- Test case: the results that research wants to use GPT-4 to create. Identify the necessary information to help a test case become effective, testers can easily use and update when there are changes in software requirements.

Our dataset is a list of use case specifications of multiple systems that we collect from various sources

The constraint of the topic is that the topic tests the use of GPT-4 for writing manual test cases to reduce the time and human dependence of this process, but does not guarantee accuracy as high as human.

2.4 Expected approach

Some related studies:

No	Research name	Citation type	Research summary	Comment
1	Automatic Creation of Acceptance Tests by Extracting Conditionals from Requirements: NLP Approach and Case Study. [1]	Journal of Systems and Software	Developed CiRA (Conditionals in Requirements Artifacts) approach to automatically derive the minimum set of required test cases from natural language requirements by applying Natural Language Processing (NLP).	Depending on the completeness of software requirements, can not produce test cases for unclear or not include condition statements. CiRA is limited to causation within a single sentence and cannot extract conditional statements that span multiple sentences.
2	Using Natural Language Processing Techniques to	ICSE-SEIP '22: Proceedings of the 44th International	Proposes an automation framework to automatically	Used to improve manual test suites. Does not support test case generation.

	Improve Manual Test Case Descriptions [2]	Conference on Software Engineering: Software Engineering in Practice	analyze test cases specified in natural language and provide useful suggestions on how to improve the test cases	
3	Automatic Generation of System Test Cases from Use Case Specifications [3]	ISSTA 2015: Proceedings of the 2015 International Symposium on Software Testing and Analysis	Proposed use case model for UMTG system test generation - an approach that automatically generates executable system test cases from use case specifications and a domain model (The domain image includes class diagrams and constraints)	Requires a lot of human intervention, not fully automated. Requires complex inputs, that follows strict regulations, and is difficult to create.
4	Automatic Generation of Acceptance Test	IEEE Transactions on Software	Introducing UMTG, an approach to automatically	Requires a lot of human intervention, not fully automated.

	Cases from Use Case Specifications: an NLP-based Approach[4]	Engineering (Volume: 48, Issue: 2, 01 February 2022)	generate executable acceptance test cases from use case specifications and domain models.	Complicated input requirements, subject to strict regulations, difficult to implement.
5	Chatting with GPT-3 for Zero-Shot Human-Like Mobile Automated GUI Testing [5]	Scientific article	GPTDroid use LLM to extract the static context of the GUI page and the dynamic context of the iterative testing process, design a prompt to enter this information into the LLM, and develop a synaptic network to decode the output of the LLM into actionable steps available to execute the application.	Use GPT and prompting techniques. Suggested steps: Start Prompt. Test Prompt. Feedback Prompt.
6	Prompting Is All You Need: Automated Android Bug	46th International Conference on Software	Propose a new approach called AdbGPT to Automatically	Use ChatGPT and prompting techniques. There are some disadvantages:

	Replay with Large Language Models [6]	Engineering (ICSE 2024)	Reproduce Errors Using LLM.	Missing steps: Do not include non-critical steps (even if they are necessary to create a test case) in the input. Component names do not match between steps and GUI component names.
7	Large Language Models are Pretty Good Zero-Shot Video Game Bug Detectors [7]	arXiv preprint arXiv:2210.02506, 2022	Explore leveraging the zero-shot and Multi-Stage Prompting capabilities of large language models to detect video game bugs.	There are still some drawbacks and the results are still full of errors.
8	Towards Transforming User Requirements to Test Cases Using MDE and NLP [8]	2019 IEEE 43rd Annual Computer Software and Applications	A meta-model approach that converts user requirements (written in different formats - both use cases and user stories) into test	Requires preparation of complex models for refining input and algorithms for conversion that are considered difficult to implement.

			cases.	
9	Autonomous Large Language Model Agents Enabling Intent-Driven Mobile GUI Testing [9]	arXiv preprint arXiv:2311.08649, 2023	Proposing DROIDAGENT, an automated GUI testing agent for Android to automate semantic, intent-driven GUI testing.	Uses GPT-4 and memory mechanisms for GPT. It is proposed to use GPT self-critique to improve the results generated by GPT.

Proposed methodology:

Prompting: Use Prompting for GPT with suitable prompting techniques such as zero-shot prompting, few-shot prompting, chain-of-thought prompting to improve output.

The topic divides the implementation process into two main phases:

Test design phase: At this phase, the main work is to extract the important information needed for creating a set of test scenarios for each case specification. At the end of this phase, GPT will filter and select the appropriate set of test scenarios that the user desires to reduce the number of test cases that need to be generated.

Test case generation phase: This phase will use the test scenario set generated in the test design phase and the information of the use case specification to produce test cases corresponding to each test scenario.

2.5 Expected result

Complete set of Agents capable of:

- Using GPT to automate the test case generation process.
- Receiving as input a set of case specifications of an application written in natural language, not constrained by a complex template.
- Returning a list of test cases to test that application with a reasonable and suitable set of test cases to test the important features.

2.6 Research plan

No	Month	Details	Results
1	9/2023	Receive topics from supervisor Read the related topic.	Understand concepts related to the topic
2	10-12/2023	Research related topics and papers.	Has knowledge about the related research and solution of these research.
3	1/2024	Research about GPT and prompting techniques for designing and improving prompts effectively to achieve better results.	Knowledge base about using GPT and related techniques.
4	2/2024	Propose method and run experiments.	Draft design of the research. Objective assessment about GPT's capabilities and having suitable adjustment.


5	3/2024	Evaluate and improve our approach to better align with the capabilities of GPT.	Propose suitable approach.
6	4-5/2024	Development and adjustment solution.	Complete approach.
7	6/2024	Run our proposal on datasets.	Assess the completion, accuracy, performance of approach.
8	7/2024	Write the thesis. Weekly testing. Update fix the error.	Thesis report

References

- [1] FISCHBACH, Jannik, et al. Automatic creation of acceptance tests by extracting conditionals from requirements: NLP approach and case study. *Journal of Systems and Software*, 2023, 197: 111549.
- [2] VIGGIATO, Markos, et al. Using natural language processing techniques to improve manual test case descriptions. In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 2022. p. 311-320.
- [3] WANG, Chunhui, et al. Automatic generation of system test cases from use case specifications. In: *Proceedings of the 2015 international symposium on software testing and analysis*. 2015. p. 385-396.
- [4] WANG, Chunhui, et al. Automatic generation of acceptance test cases from use case specifications: an nlp-based approach. *IEEE Transactions on Software Engineering*, 2020, 48.2: 585-616.
- [5] LIU, Z., et al. Chatting with GPT-3 for zero-shot human-like mobile automated GUI testing. *arXiv* 2023. *arXiv preprint arXiv:2305.09434*.
- [6] FENG, Sidong; CHEN, Chunyang. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 2024. p. 1-13.
- [7] TAESIRI, Mohammad Reza, et al. Large language models are pretty good zero-shot video game bug detectors. *arXiv preprint arXiv:2210.02506*, 2022.
- [8] ALLALA, Sai Chaithra, et al. Towards transforming user requirements to test cases using MDE and NLP. In: *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 2019. p. 350-355.


- [9] YOON, Juyeon; FELDT, Robert; YOO, Shin. Autonomous Large Language Model Agents Enabling Intent-Driven Mobile GUI Testing. arXiv preprint arXiv:2311.08649, 2023.

**SUPERVISOR
CONFIRMATION**
(Sign with full name)


Nguyễn Văn Vũ

Hồ Chí Minh City, date 16 month 07 year 2024

STUDENT GROUP
(Sign with full name)


Nguyễn Trinh Như Ý



Phạm Ngọc Anh Thư

Table of Contents

Acknowledgment	i
Table of Contents	ii
List of Figures	v
List of Tables	vii
Summary	viii
List of Acronyms	x
1 INTRODUCTION	1
1.1 Introduction	1
1.1.1 Reason choosing this topic	1
1.1.2 Main Contribution	2
1.2 Related Works	3
2 THEORETICAL BASIS	5
2.1 Software Testing Fundamentals	5
2.1.1 What is software testing?	5
2.1.2 Manual Testing and Automation testing:	5
2.1.3 Test Scenario:	8
2.1.4 Test Case:	9
2.1.5 The format of Standard Test Cases:	12

2.2	Use Case Specification:	12
2.2.1	Use Case Specification Templates:	13
2.3	Large Language Models	17
2.3.1	Prompt Engineering:	17
2.3.2	Generative Pre-trained Transformer (GPT)	18
2.3.3	Generative Pre-trained Transformer 4 (GPT 4)	19
3	PROPOSED APPROACH	20
3.1	Introduction	20
3.2	Data processing	23
3.2.1	Condition Extractor	23
3.2.2	Use case analyzer	28
3.3	Test Design	29
3.3.1	Condition Scenario Generator	29
3.3.2	Main-Flow Scenario Generator	33
3.3.3	Sub-flow Scenario Generator	35
3.3.4	Scenario Refiner	37
3.3.5	Reason for using multiple modules instead of one module	39
3.4	Test Case Generation	42
3.4.1	Test case Generator	42
3.4.2	Test Case Validator	47
4	EXPERIMENTS AND RESULTS	51
4.1	Experimental design	51
4.1.1	Dataset	51
4.1.2	Evaluation metrics	53
4.1.3	Baseline method	55
4.1.4	Procedure	56
4.2	Result	58
4.2.1	Evaluation on Overall	58
4.2.2	Evaluation on test design phase	61

4.2.3	Evaluate on test case generation phase	62
4.3	Comparison with Baseline Method	68
4.3.1	Test design phase	68
4.3.2	Test case generation phase	69
4.3.3	Comparison	69
5	CONCLUSION	74
5.1	Discussion of Experimental Results	74
5.2	Limitations	75
5.3	Future Work	76

List of Figures

3.1	Architecture overview	20
3.2	Use case example 1	23
3.3	Use case example 2	24
3.4	Example output of the Condition Extractor agent	27
3.5	Example output of the Use case Analyzer agent	29
3.6	Example input of the Condition Scenario Generator agent	31
3.7	Example output of the Condition Scenario Generator agent	32
3.8	Example Test Scenario Generated by Single Prompt	41
3.9	Example Test Cases Generated by Non-specific Test Scenario	41
3.10	Example input of the Test Case Generator agent	43
3.11	Test case format in prompt	45
3.12	Example output of the Test Case Generator agent for Scenario Password Same as Username	46
3.13	Example input of the Test Case Validator agent for Scenario Password Same as Username	47
3.14	Test case validated format in prompt	48
3.15	Example output of the Test Case Validator agent	49
4.1	Example of invalid test case with ambiguous test steps . .	54
4.2	Baseline method	56
4.3	Example of duplication still remains after refining	62
4.4	Test cases generated for scenario Navigation to a specific page of books	64

4.5	Example of incorrect test case generation leads to decrease in coverage	65
4.6	Human-Written Test Scenario Example	67
4.7	Example of a test case try to describe two separate cases .	72
4.8	Unreflected test cases for testing purchase multiple items from shopping cart	73

List of Tables

2.1	Comparison between Manual Testing and Automation Testing	7
2.2	Test Case Format	11
2.3	Reference Model of Use Case Specification	14
2.4	RUP Use Case Specification Template	15
2.5	Cockburn Use Case Specification Template	16
4.1	Total test scenarios and test cases generated	58
4.2	Accuracy of test case generated by method	59
4.3	Coverage evaluation	60
4.4	Accuracy of test scenario generated in test design phase . .	61
4.5	Test case generation result	62
4.6	Accuracy evaluation on new test case set	63
4.7	Coverage evaluation on new test case set	64
4.8	Accuracy evaluation on test case set using human-written test scenario	67
4.9	Total test scenarios and test cases generated from two methods	69
4.10	Accuracy of test scenario generation	70
4.11	Accuracy of test case generation	71

Summary

In the software testing process, generating manual test cases is a very important task. Although automated testing has developed fast and become a trend, for complex functions, writing automated test cases could be challenging. Additionally, there is a need for manual test cases to align with requirements from important customer or company documents for easier management. Therefore, writing test cases in natural language is still necessary.

However, writing a manual test suite requires a lot of human effort and time. The cost that companies and organizations have to pay for this progress is also a big concern. Automating the generation of manual test cases is a topic that has been researched for a long time but has not yet been widely applied due to strict requirements or formats on software requirements or use case specifications.

With the appearances of large language models such as BERT and GPT, we hope to resolve this problem by using GPT-4 model to process the input and automatically generate manual test cases without having to set many constraints for the input.

In this work, we propose a method to use GPT-4 for automatically generating manual test cases and evaluate the results. The research aims to find the strengths and weaknesses of the proposed method based on various datasets. Our goal for this research is to propose a method that helps to utilize GPT-4 more effectively in generating manual test cases based on use case specifications.

Our proposed method is based on the divide-and-conquer idea. We divide the important information from the use case specifications into smaller parts and use GPT-4 to process them separately. We also use the self refining method where GPT-4 self-evaluates its results to filter out incorrect or redundant outputs. Finally, we compile all the generated test cases into a complete test suite for a function.

Our results show that our propose is capable of generating manual test cases based on functional specification information. However, this propose still encounters several issues, such as duplicate test cases, unclear test cases, and generating test cases that cannot be executed. These problems add work for responsible individuals who need to filter out unnecessary or unusable test cases. We also compared the proposed method with the results obtained from simulating the human simple interaction process with GPT to create test cases. According to the comparison results, our proposed method reduces the amount of work humans need to do after receiving the results.

List of acronyms

#	Abbreviation	Full Form
1	GPT	Generative Pre-trained Transformer
2	LLM	Large Language Model
3	TS	Test Scenario
4	TC	Test Case
5	SF	Scenario Found
6	SC	Scenario Cover
7	BC	Book Catalog Website
8	EL	English Learning Website
9	HM	Hotel Management System
10	OS	Online Shopping System
11	UC	Use Case Specification
12	NLP	Natural Language Processing
13	BA	Business Analyst
14	QA	Quality Assurance
15	RUP	Rational Unified Process
16	BERT	Bidirectional Encoder Representations from Transformers
17	SRS	Software Requirements Specification
18	RUCM	Restricted Use Case Modeling

Chapter 1

INTRODUCTION

1.1 Introduction

1.1.1 Reason choosing this topic

Software testing is an essential phase in the software development process, helping developers to ensure that software meets all requirements. Despite advances in methods and tools for test automation, manual testing still plays an important role in the industry with the ability to handle complex, logic-rich, and frequently changing requirements that automation tools often struggle with.

Manual test cases written in natural language are accessible to a wide range of stakeholders, including testers, developers, project managers and who may not have deep knowledge in software testing. Many organizations still rely on manual test case creation despite advancements in automated testing tools. Manual test cases could help to explore complex functions that are hard to write test script for automation.

But writing manual test cases from use case specifications requires huge efforts from human. This progress including read through out every use case specifications, finding important information and conduct relevant manual test cases, usually done manually by human.

With the development of LLMs for natural language related tasks, we

want to automatic this progress. Large Language Models (LLMs) for automating manual test case creation holds huge promise. Models like GPT-4, excel in natural language understanding and generation tasks, making them ideal candidates for assisting or even autonomously generating test cases from use case specifications. By automating manual test case creation, organizations can reduce time and cost needed for software testing.

This study proposes an approach to automatically generating test cases for manual testing from requirements in forms of use case specifications. Our approach leverages capabilities of large language models (LLM) for natural language processing and logical reasoning to analyze requirements and generate test cases

1.1.2 Main Contribution

Our main contributions include:

- An approach receives input as a set of use case specifications and automates the process of identification, and separation key use case sections, including flow transitions, conditions, and components, to create and refine test scenarios. The generated test scenarios are then used to generate corresponding test cases, which are feedback for accuracy and reflection to ensure corresponding and comprehensive coverage of all use case flows and conditions.
- An empirical examination of our proposal to generate test scenarios and test cases for four applications and a thorough evaluation of the effectiveness of test cases created between our propose and baseline method. Our approach generates 308 test scenarios and 427 test cases for 44 use cases with 90.61% flow coverage and 97.20% condition coverage accuracy. Our approach achieves high coverage of use cases while producing significantly fewer test scenarios and test cases compared to the baseline.

- Conduct experiments to evaluate the capabilities of our method in capturing key information, identifying important scenarios within use cases, and generating corresponding test cases. These experiments aim to determine the effectiveness of our method in automating the process of test case generation based on use case descriptions.

1.2 Related Works

Software testing is an important process to evaluate the quality of a software product, reduce maintenance cost and resolve problem. It is essential to the life cycle of software development. Automatic test case generation has been researched extensively to minimize human effort and improve the efficiency of this vital process.

Trending topics in this field are automatic creation of test scripts for automate testing. Many recently surveys [1]–[3] on LLMs and Software Testing have proved that there are few research on transforming use case specification to manual test cases.

Wang et al [4] presented a technique that uses an NLP pipeline to automatically create system test cases from RUCM use case specifications — specifications that are expressed in a strictly manner. Beside the RUCM specifications, tester or anyone who is responsible for this work also have to prepare a domain model for this approach. To evaluate their technique, their study conducted a case study on a real-world system called BodySense comes from IEE Smart Sensing Solutions. They created test cases for the BodySense system and then had the testers working on BodySense evaluate the results.

Efforts have been made to decrease the work required to design inputs. Gropler et al [5] propose a semi-automated technique that utilizes an NLP parser to reduce human effort by automatically generating test cases.

Fischbach et al [6] have conducted research on using BERT to extract conditional requirements for automatic creation of acceptance test case

based on conditional statements written in user’s requirements or user story. But there are challenges in detecting NL conditional statements since natural language is complex and rich, therefore affects the accuracy of the method. Their approach starts by using BERT to find all the conditional statements in the text. After extracting these statements, they use BERT again to label the words in each statement. These labels are then transformed into Cause-Effect-Graphs, with each graph representing one statement. From these graphs, they derive the test cases needed for acceptance testing.

Recently, Rahman et al [7] presented a research on employing GPT-4 to generate user stories and their corresponding test cases from software requirements documents.

With the development of pre-trained LLMs, many tasks in software engineering could be automatically done with high potential, including software testing. However, despite the recent efforts to maximize the potential of LLMs for generating test cases, there has been relatively little work focused on utilizing LLMs for generating manual test cases comparing to other fields of testing.

Our approach aims to automatically generate manual test cases directly from use case specifications in many forms with GPT-4 using prompt engineering.

Chapter 2

THEORETICAL BASIS

2.1 Software Testing Fundamentals

2.1.1 What is software testing?

Software testing is an essential method in the software development lifecycle [8]. It is used to check whether the software product meets all expected requirements and is defect-free. This process involves the execution of software/system components by operating software manually or using automated tools to evaluate product perfection. The main objective of software testing is to identify defects, gaps, or missing requirements compared to the actual requirements, ensuring the product's quality.

2.1.2 Manual Testing and Automation testing:

Manual testing is a process where human testers execute test cases without the assistance of automation tools. The main objective is to identify defects, bugs, and issues within the software application. Test Suites or cases, are designed during the testing phase and should have 100% test coverage. This approach can be used in various techniques:

- Black Box Testing

- White Box Testing
- Unit Testing
- System Testing
- Integration Testing
- Acceptance Testing

Manual testing is ideal for complex and non-repetitive scenarios, making investing in automation time-consuming. One another situation is when a tester wants to evaluate the application from the end user's perspective, providing more authentic and human feedback to the development team.

Furthermore, manual testing is crucial for new applications to ensure they are error-free and meet the specified requirements. It requires significant effort but is vital for applications where human judgment is necessary.

Automation testing uses software tools and scripts to perform tests on the software automatically. The primary purpose is to execute test steps, compare actual and expected results, and generate test reports without human intervention. Automation testing is highly effective for repetitive, time-consuming tasks and is ideal for regression, load, and performance testing. It ensures higher accuracy, speed, and reliability but requires an initial investment in tools and programming skills.

Test cases suitable for automation are:

- Test cases that are crucial to the business and pose high risks.
- Test cases that need to be executed frequently.
- Test cases that require a lot of time to execute manually.
- Test Cases that are very tedious or difficult to perform manually

Test cases are not suitable for automation:

- Test cases that haven't been executed manually even once.
- Test cases where the requirements often change.
- Test cases which are executed on an ad-hoc basis

Criteria	Manual Testing	Automation Testing
Reliability	More prone to human error	More reliable, consistent results
Performance Testing	Not possible	Possible
Exploratory Testing	Possible	Not possible
Human Observation	Yes	No
Parallel Execution	Possible but increasing human resources	Possible by using different platforms
Response to UI Change	Easily adaptable	Requires re-scripting
Test Report Visibility	Recorded in an Excel or Word, not readily available	Stored in automation system, can re-display
Set up	No setup required	Requires setup
Programming knowledge	Not required	Required
Processing time	Slower	Faster
Investment	Less initial cost	Higher initial cost
Initial Investment	Low	High
Ideal approach	Only run once or twice test case set	Frequently executing test cases set
When to Use	Ad-hoc, exploratory, and usability testing	Regression, load, and performance testing

Table 2.1: Comparison between Manual Testing and Automation Testing

Key Differences between Manual and Automation Testing Manual and Automation testing are two essential methods in software quality

assurance, each with unique advantages and limitations. Manual testing relies on human skills and intuition to uncover issues, while Automation testing utilizes software tools to execute predefined tests efficiently. The Table 2.1 compares key criteria between manual and Automation testing.

2.1.3 Test Scenario:

Test scenario, also called Test Condition or Test Possibility, is a high-level description refers to any functionality or feature of the software application that can be tested. It's about thinking from the end user's perspective and determining real-world scenarios to validate the application. It outlines a specific situation or use case that the software should be able to handle.

The need of Test Scenarios:

Test Scenarios are created for the multiple reasons, some of them that can be listed are:

- Ensures the fully test coverage.
- Can be approved by stakeholders to confirm the software meets ideal use cases.
- Helps plan and organize testing efforts to set the priority of use cases.
- Essential for studying the software's end-to-end functionality.

How to Write Test Scenarios

Testers should follow these five steps to create Test Scenarios:

- Read requirement documents like business requirement specification (BRS), software requirement specification (SRS), functional requirement specification (FRS) of the System Under Test (SUT)

- For each requirement, determine possible user actions and objectives.
- List different test scenarios for each software feature.
- Create a traceability matrix to ensure all requirements have corresponding scenarios.
- Review scenarios with supervisors and stakeholders.

2.1.4 Test Case:

A test case is a fundamental component in the software testing process. It is a detailed document that describes a specific flow, condition or set of conditions under which a software application or system is tested. The primary goal of a test case is to ensure that the software behaves as expected in various scenarios, which are derived from the requirements and use cases of the system. A well-defined test case includes test steps, execution conditions, and expected results, providing a systematic way to validate the software's functionality. Test cases are usually created by the QA or testing team and serve as detailed, step-by-step guidelines for performing system tests. Testing starts after the development team completes a system feature or a set of features. A group of related test cases is called a test suite.

Characteristics of good Test Cases

When designing effective test cases, there are several key principles to design a good test cases set:

- **Simplicity and Clarity:** Test cases should be as straightforward as possible. It's important to write them in a clear and unambiguous manner because the person executing the test may not have been involved in creating it. Using direct commands like “navigate to the

homepage,” “enter data,” or “click the button” helps make the test steps easy to follow and speeds up the execution process.

- Think like end user: The main aim of creating test cases is to ensure that they reflect the needs and expectations of the end user. Test cases should be crafted from the perspective of the user to ensure that the software meets the customer’s requirements and is user-friendly.
- Make No Assumptions: Test cases should be based on explicit specifications rather than assumptions. Always refer to the Specification Documents to define what needs to be tested, ensuring that testers are not overlooking or misinterpreting features.
- Ensuring Comprehensive Coverage: Test cases should aim to cover all requirements detailed in the specification document. Using a Traceability Matrix can help verify that all functions and conditions are tested, leaving no requirement unexamined.
- Clear Identification: Each test case should have a unique and identifiable ID to facilitate tracking and defect management. A well-named test case ID makes it easier to refer to specific tests when reporting issues or reviewing requirements.
- Applying Testing Techniques: Given that it’s not feasible to test every possible scenario, employing established testing techniques can be very beneficial. Techniques like Boundary Value Analysis (BVA), Equivalence Partitioning (EP), State Transition Testing, and Error Guessing are useful methods for identifying significant test cases and potential defects.
- Repeatability and Independence: A well-designed test case should produce the same results every time it is executed, regardless of who performs the test. This consistency ensures that test outcomes are reliable and the test cases are effective.

Field	Description
Test Case ID	A unique identifier for the test case.
Title/Description	A concise description of the purpose of the test case.
Test Objective	The specific goal or objective of the test.
Preconditions	Any necessary conditions that must be met before the test is executed.
Test Steps	A step-by-step sequence of actions to perform during the test.
Input Data	The data or parameters to be used as input for the test.
Expected Results	The anticipated outcomes or behaviors after executing the test steps.
Actual Results	The actual outcomes observed when executing the test.
Test Environment	Details about the system, hardware, software, and configurations used for testing.
Test Data Preparation	Instructions on how to set up the required test data.
Test Execution Date	The date and time when the test was executed.
Test Execution Status	The pass/fail status of the test case after execution.
Test Conclusion	A summary of the results and observations of the test.
Attachments	Any relevant files, screenshots, or documentation associated with the test.
Test Case Author	The person responsible for creating the test case.
Test Case Reviewer	The person who reviewed and approved the test case.
Test Case Version	The version or revision number of the test case.
Notes/Comments	Additional information, insights, or comments related to the test case.

Table 2.2: Test Case Format

2.1.5 The format of Standard Test Cases:

In the software development industry, each firm often develops its own test case templates that fit their team's unique workflows and requirements. This customization ensures that the test cases align well with the specific needs and priorities of their projects. The following format in Table 2.2 serves as a reference model, providing a simplified view of the test case.

2.2 Use Case Specification:

A use case specification is a detailed document that describes the interactions between an actor (a user or other system) and the system under considered to achieve a specific goal. This document outlines the functional requirements of the system by specifying the sequence of actions and events that determine the system's behavior under various conditions. The use case specification serves as a detailed plan for designing and implementing a software system, ensuring that all functional requirements are clearly captured and understood.

The primary purpose of a use case specification is to capture the functional requirements of a system in a structured and understandable manner. This ensures that all stakeholders, including developers, testers, and business analysts, have a clear and consistent understanding of how the system should behave under various conditions. The use case specification serves several important purposes:

- **Requirements Elicitation:** Use case specifications help in identifying and documenting the functional requirements of a system. By describing the interactions between actors and the system, they ensure that all necessary functionalities are all captured and understood.
- **Communication:** Use case specifications provide a software description that is easy to understand for all stakeholders (both technical

and non-technical ones) involved in the development process, ensuring that everyone has a clear understanding of the system’s requirements and behavior.

- **Design and Implementation:** Use case specifications serve as a detailed plan for designing and implementing the system, guiding developers in understanding the expected system behavior and helping in creating a design that meets the specified requirements.
- **Testing and Validation:** Use case specifications are a valuable resource for creating test cases and validating the system. They provide a basis for identifying test scenarios, designing test cases, and ensuring that the system meets its functional requirements.

2.2.1 Use Case Specification Templates:

In the software development industry, each company often creates its own use case specification template with different components tailored to their team’s specific workflows and requirements. This customization ensures that the template meets the unique needs and priorities of their projects. The following format in Table 2.3 serves as a reference model, illustrating a simplified overview of the common components typically found in a use case specification. These components form the basis for crafting detailed and effective use case documents

Two prominent templates that have been widely adopted are the RUP (Rational Unified Process) [9] Template and the Cockburn Template [10]. Despite their origins in earlier years, these templates continue to be relevant and provide a structured approach to capturing use case specifications.

Component	Description
Use Case Title	Provide a concise name for the use case.
Use Case ID	Assign a unique identifier for the use case for reference and tracking.
Actors	List the actors involved in the use case (e.g., users, systems, devices).
Goal	Define the objective or desired outcome that the actor aims to achieve.
Preconditions	Specify the conditions that must be true before the use case can start.
Postconditions	State the conditions that will be true after the use case has successfully completed.
Triggers	Identify the events that initiate the use case.
Main Flow	Describe the primary sequence of steps or interactions between the actor and the system that lead to the goal.
Alternative Flows	Provide variations or alternative sequences of steps that still achieve the goal under different conditions.
Exception Flows	Outline the paths that describe what happens when an error or unexpected condition occurs, preventing the goal from being achieved.
Assumptions	Document any underlying assumptions that affect the use case.
Special Requirements	Capture any additional requirements or constraints that are specific to the use case (e.g., performance criteria, regulatory constraints).
References	List any other documents, use cases, or external references that provide additional context or information.

Table 2.3: Reference Model of Use Case Specification

RUP Template

The RUP template¹, developed by Rational Software (now part of IBM Application Management Services), is an iterative software development process framework. It includes specific guidelines and templates for documenting use cases. The RUP template is characterized by its structured and systematic approach, making it suitable for comprehensive documentation and traceability. The template [9] typically includes the following sections shown in Table 2.4:

Component	Description
Use Case Name and ID	A unique identifier and a clear name for the use case.
Description	A brief overview of the use case, including its purpose and objectives.
Actors	The list of actors involved in the use case.
Preconditions	Conditions that must be met before the use case can begin.
Post-conditions	Conditions that will be true after the successful completion of the use case.
Main Success Scenario	The primary sequence of steps that lead to the successful completion of the use case, reach the use case goal.
Extensions	Variations, alternative flows, or exceptional flows that may happen in the execution process of the use case.
Special Requirements	Any additional requirements or constraints specific to the use case.
Notes and Issues	Additional comments, notes, or issues related to the use case.
References	List of links to related documents or resources.

Table 2.4: RUP Use Case Specification Template

¹https://files.defcon.no/RUP/webtmpl/templates/req/rup_ucspect.htm

Cockburn Template

The Cockburn template, developed by Alistair Cockburn, is another widely used format for documenting use cases. It focuses on capturing essential information in a concise and structured manner, emphasizing readability and understanding. The Cockburn template [10] typically includes the following sections Table 2.5:

Component	Description
Use Case Name	A clear and concise name summarizing the functionality the use case represents.
Scope	The system boundary; what the system will cover.
Level	Indicates whether the use case is a user goal or a subfunction.
Primary Actor	The main entity (user or system) that interacts with the system.
Stakeholders and Interests	Other parties interested in the outcome and their concerns.
Preconditions	The state of the system before the use case starts.
Postconditions	The state of the system after the use case completes successfully.
Main Success Scenario	A step-by-step description of how the primary actor achieves their goal without any interruptions.
Extensions	Variations from the main success scenario, including alternate paths and exceptions.
Special Requirements	Non-functional requirements, such as performance or security constraints.
Technology and Data Variations	Any variations in technology or data that could affect the use case.
Frequency of Use	How often the use case is expected to be executed.
Miscellaneous	Any additional information that does not fit into the other categories.

Table 2.5: Cockburn Use Case Specification Template

2.3 Large Language Models

Large Language Model (LLM) refers to an Artificial Intelligence (AI) model which is trained on a large scale of data to have the capability to perform relating human natural language tasks with the using of prompting engineering. As auto-regressive models, the training of LLMs is taking an input text and repeatedly predicting the next token or word.

LLM is based on deep learning techniques (for example transformers) have the ability to recognize and operate on natural language, involving: generating text, answering question, translation, content summarizing, and analysing sentiment.

At the beginning of the emergence of LLMs, fine tuning was the only way make a LLM could be adapt to specific tasks. Recent LLMs, such as GPT, LLaMA [11] and PaLM [12], however, can achieve similar task-specific results without requiring additional training through prompt-engineering.

2.3.1 Prompt Engineering:

Prompt engineering is the process of formulating text prompts that effectively communicate and describe tasks to Large Language Models (LLMs). Prompt engineering focuses on creating and refining prompts to maximize the efficiency and effectiveness of LLMs across diverse applications and research areas.

The discipline of prompt engineering is essential for enhancing the performance of LLMs in various tasks, ranging from simple question answering to more complex arithmetic reasoning. This knowledge helps in exploiting the full potential of LLMs and recognizing their limitations.

There are various type of prompting techniques that is used to effectively design and improve prompts to get better results on different tasks with LLMs. Some are commonly known is:

- Zero-shot prompting: the prompt used to interact with the model

won't contain examples or demonstrations. The zero-shot prompt directly instructs the model to perform a task without any additional examples to guide it.

- Few-shot prompting: technique to enable in-context learning where engineers provide demonstrations in the prompt to guide the model to better performance.
- Chain-of-Thought prompting: enables complex reasoning through providing examples and describing the intermediate reasoning steps that lead to the desired output.

2.3.2 Generative Pre-trained Transformer (GPT)

Generative Pre-trained Transformer [13], developed by OpenAI. GPT is a family of large language models (LLMs) that can understand and generate text in natural language.

- Generative: Capable of producing content such as text and images.
- Pre-trained: Models that have been trained on large datasets to perform specific tasks.
- Transformer: A deep learning architecture that converts input data into different types of output.

GPT models are computer programs that serve as general-purpose language prediction models, able to analyze, summarize, and otherwise use information to generate content. GPT has seen three major iterations since its first researched in 2018 [14]: GPT-2[15], GPT-3 [16], and GPT-4[13].

There are researches that compare GPT with other models [17]–[19], we find that GPT could do multi-task without the need of fine-tuning. We choose GPT as our model to use in our propose.

2.3.3 Generative Pre-trained Transformer 4 (GPT 4)

GPT-4 is the latest released model of GPT, it has ability of parsing image inputs as well as text. Currently, GPT-4 achieve human-level performance on various professional and academic benchmarks. There has been technical report [13], [20] comparing performance of GPT-3.5 and GPT-4, as GPT-4 is proved to be better, we decide to use this for our propose.

Chapter 3

PROPOSED APPROACH

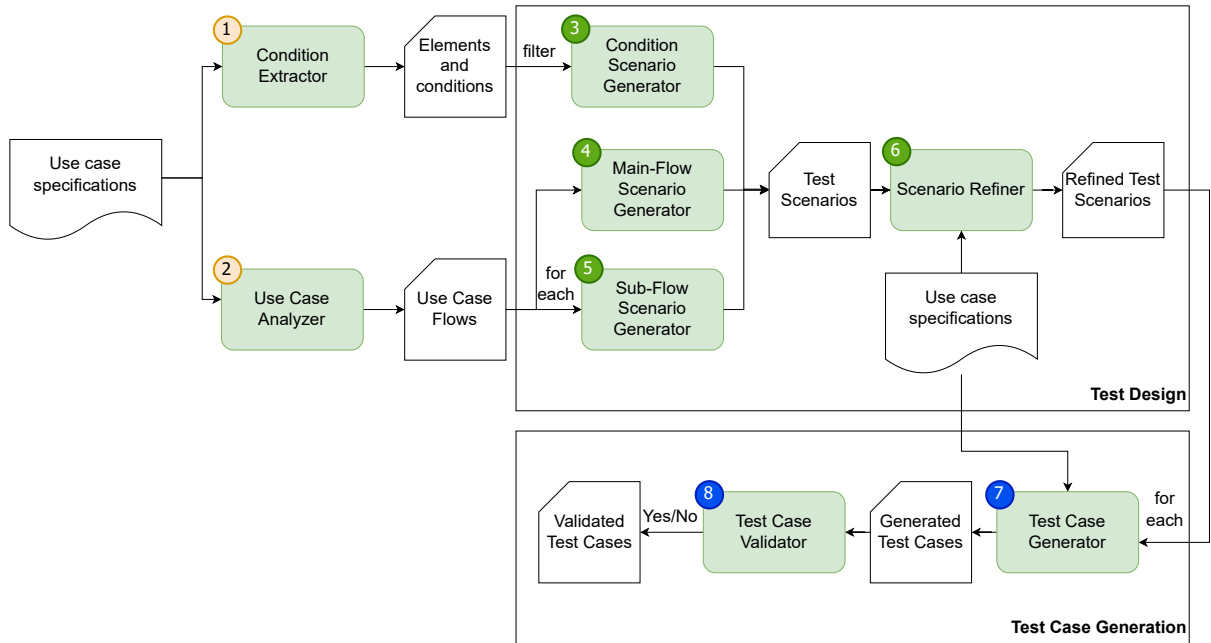


Figure 3.1: Architecture overview

3.1 Introduction

Our proposed approach is illustrated as Figure 3.1. We divided the generation of manual test cases process into 2 main phases: Test Design and Test Case Generation. The idea to split the process into 2 phases of

this approach is inspired by the vision of Heumann [21] , which emphasizes the importance of identifying all scenarios that need to be tested before generating test cases. Our method includes 8 modules, each leveraging GPT-4 and using zero-shot prompting technique. We have a total of 4 modules involved in Test Design phase and 2 modules responsible for Test Case Generation. Additionally, two modules handle input processing to ensure that the input data is properly segmented and extracted and fed into suitable modules.

1. **Test design phase:** This phase reads all the conditions, behaviors, and cases detailed in the use case specifications, which are extracted during data processing that need to be tested to ensure the application behaves as described and is error-free. The goal is to design a set of test scenarios that optimize coverage of conditions and cases for each use case. To achieve this, we have developed components that prioritize specific targets:
 - Condition Coverage: Condition Scenario Generator module analyzes the extracted interaction elements and their conditions extracted by the Condition Extractor module. It then generates test scenarios for these elements and conditions, ensuring comprehensive coverage and no condition is missed.
 - Flow coverage: The Main-Flow Scenario Generator and Sub-Flow Scenario Generator modules ensure that all action flows described in the use case specifications are considered. These modules use the results from the Use Case Analyzer to generate test scenarios that cover every possible flow.
 - The generated test scenarios from above components are then refined by Scenario Refiner module that eliminates unnecessary or duplicate scenarios, resulting in a final set of test scenarios optimized for coverage and relevance.

2. **Test cases generation phase:** The function of this phase is to use the results of the designed scenarios of the previous phase and detailed descriptions of use cases to create corresponding test cases. For each test scenario, a module generates the corresponding test cases and another one reviews them to ensure they accurately reflect the given test scenario, producing a final validated set of test cases. Each test case includes information such as the test case goal, test steps, and expected results.

In reality, businesses, organizations and even in schools and teaching, people who play the role of Business Analyst can have many ways of writing use cases for their applications, resulted in vary widely format of use case specifications. While there are some use case sets that have a sketchy format and lack information, there are also specific use case sets with full detailed descriptions that often have a lot of information leading to miss information during processing. Because of this, Our application aims to solve this problem by partitioning this complex content into more simple segments, enabling each component to accurately understand the context of each piece of information.

With our proposal, we do not generate test data for the generated test cases for the reason that our input data comes in diverse formats from various sources, while some data has simple formatting and non-specific information, such as missing information about which fields to enter, which makes it difficult for our method to detect and generate the corresponding test data. This can lead our method to generate test steps that are not described in the use case, resulting in test cases that are not practical for testing purposes.

```

1 Use case name: Purchase
2 Description: This feature allows users to purchase of items they have added to their
  shopping cart or at product detail page.
3 Actor: User
4 Preconditions: None
5 Postconditions: user can checkout order.
6 Basic Flow:
7     Step 1: User in the shopping cart page and User has added items to the shopping cart.
8     Step 2: User select items to checkout by selecting each items by clicking on checkbox
  before item.
9     Step 3: System displays the summary costs.
10    Step 4: User click the "Checkout" button.
11    Step 5: System process to checkout.
12        - User is redirected to page that shows one or many new orders of all the items
  user have chosen.
13        - Products from different shops will be separate into different orders, products
  from the same shop is in one order.
14
15 Alternative flow:
16     1. In shopping cart page: User can select all the items come from one store, user
  choose by clicking the checkbox at the head of the store.
17     2. User can purchase products in product detail page:
18         - User perform feature 'View a product detail'.
19         - User click button 'Buy Now'.
20         - If product has many options, user choose one available option before adding
  product to cart.
21         - User can adjust quantity of product before adding product into cart by clicking
  on minus or plus button next to quantity of the product.
22         - User click the "Checkout" button.
23         - System process to checkout.
24         - User is redirected to page that shows one order of the item user has chosen.
25
26 Exception flow:
27     - user cannot click on the checkbox of the product that is out of stock or erased by
  the seller even though it is in user's shopping cart.
28 When user purchase products in product detail page:
29     - user cannot purchase product has many options without selecting one available
  option.
30     - user cannot purchase product with quantity that is bigger than the current stock or
  lower than one.
31     - User cannot purchase product with no stock or out of stock option of one product
  with multiple options.
32     - the button "Checkout" is disable if the selected product is invalid

```

Figure 3.2: Use case example 1

3.2 Data processing

3.2.1 Condition Extractor

Condition Extractor module (Block 1 Figure 3.1) is designed to analyze the use case specification to identify interactive elements and their associated conditions. This module ensures that all conditions that need to be tested are captured accurately, allowing for comprehensive test coverage.

This module's idea come from S. Feng and C. Chen research[22] with

```

1 Use case name: Registry
2 Actor: Learner
3 Precondition: Learner is in Registry page.
4 Main flow:
5     Step 1: Learner fills in the username field by a valid username, a valid username
        must be over 8 characters and below 30 characters.
6     Step 2: Learner fills in the password field by a valid password, a valid password
        must be over 8 characters and below 30 characters, contains at least one of each kind
        : a normal character, a capitalize character, a number and a special character. A
        valid password cannot be the same as the username.
7     Step 3: Learner press "Registry" button
8     Step 4: System redirects learner to Home page
9 Alternative flow 1: Learner register by email
10    At step 1 of the basic flow: Learner clicks on the Mail icon
11    Step 2: Learner fills in the email field by a valid email, a valid email must be an
        exist email with the right format
12    Go back to step 2 in the basic flow and continue with the steps from step 2.
13
14 Exception flow 1: Learner enters invalid username
15    At step 1 of the basic flow: Learner fills in the username field by an invalid
        username
16    Go back to step 2 in the basic flow and continue with the steps from step 2.
17    At step 4 of the basic flow: System shows a notification that the username is invalid
18
19 Exception flow 2: Learner enters invalid password
20    At step 2 of the basic flow: Learner fills in the password field by an invalid
        password
21    Go back to step 3 in the basic flow and take only step 3.
22    At step 4 of the basic flow: System shows a notification that the password is invalid
23
24 Exception flow 3: Learner enters an existed username
25    At step 1 of the basic flow: Learner fills in the username field by an username that
        has been registered
26    Go back to step 2 in the basic flow and continue with the steps from step 2.
27    At step 4 of the basic flow: System shows a notification that this username has been
        registered and ask the learner to choose another username.
28
29 Exception flow 4: Learner enters invalid email
30    At step 2 of the alternative flow 1: Learner fills in the email field by an invalid
        email, an invalid email must be an inexist email or email with the wrong format
31    Go back to step 3 in the alternative flow 1 and continue with the steps from step 3.
32    At step 4 of the basic flow: System shows a notification that the mail is invalid
33 Postcondition: Regist successfully

```

Figure 3.3: Use case example 2

'S2R Entity Extraction' when that research acknowledges the same problem that is the propose need an in-depth understanding of the potential entities presented in the input and one action can be represented in multiple formats. Using the input directly can make it difficult to identify these entities and often lose information of action or produces inappropriate results. They designed 'S2R Entity Extraction' to address this. Similarly, our approach also suffers from the same problem of various elements and their conditions being presented in use cases, using use cases directly to

generate test cases can lead to omitting important elements or conditions. For example, when implementing the login function of a banking application, developers often require many conditions (e.g username must have more than 8 characters, username must have at least one special character, username must have at least one number, and so on) for the login username to ensure security. These conditions are sometimes missed if use cases are used directly to generate test cases.

Similar to the above research, we identify fifteen standard element types: Button, Icon, Scroll, Text Field, Tab, Radio Buttons, Menu, Combo Box, Slider, Switch, Dialog, Link, Form, Rating and Filter.

These elements represent a comprehensive set of interactive objects in application design. However, in this paper, we focus on the most frequently used type with multiple criteria: text fields. Other element types can be considered in future studies.

Besides the type of element, this component also extracts conditions that determine elements validity or invalidity. This process allows us to fully capture the conditions mentioned in the use case to ensure comprehensive condition coverage. In addition, determining the type of attribute also allows us to conveniently filter elements based on element type so that we can develop and design further components to generate test scenarios suitable for each type. It is also helpful to exclude elements that do not need to generate test scenarios like button or tab that just for clicking and also exclude elements that don't have any conditions to test, helping Condition Scenario Generator module easily identify which elements are truly need to be generated test scenarios, solving the problem that module tends to create many test scenarios to verify the accessibility of each element mentioned in the use case, leading to many redundant test scenarios.

To ensure accurate extraction that easily groups objects with the same test as well as prevent the creation of unnecessary conditions, the following prompting rules are used:

- **Do Not Arbitrarily Create Conditions:** The component must only extract conditions that are explicitly mentioned in the use case flow. Do not create additional conditions or assumptions that are not stated in the use case, thereby limiting subsequent modules from creating unnecessary test scenarios.
- **Consistency in Element Types:** The module must consistently categorize elements into one of the predefined types, even if the use case description is vague or incomplete. This is to ensure that filtering process of elements works effectively.

The output of the Condition Extractor module is a JSON object that includes:

- The name of each interactive element.
- The conditions that make the element valid and invalid.
- The type of the element.

An example of this Condition Extractor module output of the use case Login in Figure 3.3 is illustrated in Figure 3.4

However, several challenges were encountered during the development and implementation of this module. One significant challenge was ensuring the accurate extraction of all relevant conditions, especially when the use case specifications were insufficiently detailed or vague. In many instances, use case descriptions merely include the names of elements and basic actions such as "touch," "click," or "press," without specifying the type of element. This lack of specificity can lead to ambiguities where this module must infer whether an element can be: a button, radio button, combo box, checkbox, or another type. Defining the wrong element type can lead to creating wrong scenarios in the next step and that wrong scenarios have no test value. While module's predictions are generally accurate, there are instances where incorrect predictions lead to errors in scenario design.

```

1 {
2   "Username": {
3     "condition": {
4       "valid": "must be over 8 characters and below 30 characters, must be entered",
5       "invalid": "below 8 characters or over 30 characters, has been registered"
6     },
7     "type": "text field"
8   },
9   "Password": {
10    "condition": {
11      "valid": "must be over 8 characters and below 30 characters, contains at least one
12      of each kind: a normal character, a capitalize character, a number and a special
13      character, cannot be the same as the username",
14      "invalid": "below 8 characters or over 30 characters, does not contain at least one
15      of each kind: a normal character, a capitalize character, a number and a special
16      character, is the same as the username"
17    },
18    "type": "text field"
19  },
20  "Registry button": {
21    "condition": {
22      "valid": "",
23      "invalid": ""
24    },
25    "type": "button"
26  },
27  "Mail icon": {
28    "condition": {
29      "valid": "",
30      "invalid": ""
31    },
32    "type": "icon"
33  },
34  "Email": {
35    "condition": {
36      "valid": "must be an exist email with the right format",
37      "invalid": "an inexist email or email with the wrong format"
38    },
39    "type": "text field"
40  }
41 }

```

Figure 3.4: Example output of the Condition Extractor agent

Currently, this component focuses on text fields as the primary element type. Future work could expand the extraction to include other element types. Additionally, refining the condition extraction process to handle more complex conditions and relationships between elements could further improve the module's performance.

The Condition Extractor module plays a critical role in our approach by ensuring that all interactive elements and their conditions are thoroughly extracted from use case specifications.

3.2.2 Use case analyzer

The Use Case Analyzer module (Block 2, Figure 3.1) is an integral part of our approach, designed to analyze and extract essential flow details from use case specifications. This component ensures that only actionable process steps are retained, excluding unnecessary information such as use case goals, assumptions, summary, stakeholders and interests and so on. By focusing only on activity flows, this component prepares the use case for the next stages of our approach, facilitating the creation of reasonable test scenarios.

The primary purpose of the Use Case Analyzer module is to streamline the process of converting varied and complex use case specifications into a standardized format that can be easily processed by subsequent modules. By extracting only the essential flow details and excluding irrelevant information, this component ensures that the input for the Main-Flow Scenario Generator and Sub-Flow Scenario Generator is clear, concise, and relevant.

Since the input provided by users can vary greatly in format, as illustrated in Figure 3.2, Figure 3.3. Given the diversity in input formats due to different company terminologies and templates, it is hard to give out a strictly specific format, as this would require users to modify their original formats. Instead, by using this component, users can simply describe their use case specifications in their usual format with flows, which can follow the Cockburn template [10], the RUP template [9], or other similar templates. The primary task of this component is to detect the number of flows within the use case, identify the Main-Flow, and group each flow with its corresponding details.

The process begins with this component receiving the use case input. It then parses the input to identify different flows and skips non-actionable information such as descriptions, preconditions, and postconditions, focusing solely on the flow actions. The output of the Use case Analyzer are extracted flows that formatted into JSON blocks. The format JSON allow

us to easily separate and operate on them. An example of the output of Use case Analyzer is illustrates in figure 3.5

```

1 {
2   "Main flow": [
3     "Step 1: Learner fills in the username field by a valid username that has been
      registered",
4     "Step 2: Learner fills in the password field by the correct password for the
      corresponding username",
5     "Step 3: Learner presses \"Login\" button",
6     "Step 4: System redirects learner to Home page"
7   ],
8   "Alternative flow: Login by email": [
9     "At step 1 of the main flow: Learner fills in the username field by a valid email
      that has been registered",
10    "Step 2: Learner fills in the password field by the correct password for the
      corresponding email",
11    "Go back to step 3 in the main flow and continue with the steps from step 3"
12  ]
13 }

```

Figure 3.5: Example output of the Use case Analyzer agent

The use case analyzer is equipped with mechanisms to handle poorly formatted use cases. In addition, dividing information into parts and sending to corresponding next modules also helps these modules receive appropriate information and thereby have appropriate commands and instructions for generating proper test scenarios.

In conclusion, by focusing solely on actionable flow details and discarding ancillary information, the Use Case Analyzer ensures that downstream components, such as the Main-Flow Scenario Generator and Sub-Flow Scenario Generator, receive clear and relevant input. This streamlined approach enhances the accuracy of scenario generation by eliminating ambiguity and irrelevant data points from the outset.

3.3 Test Design

3.3.1 Condition Scenario Generator

The Condition Scenario Generator module (Block 3 Figure 3.1) is designed to generate test scenarios that test all the conditions for text field

elements mentioned in the use case under consideration.

This component's idea come from the idea of Fischbach et al's research [6]. In their study, they explored the potential of automatically generating the minimal set of required test cases from conditional statements in informal requirements using NLP. This research focus on creating acceptance test cases for conditional statements only. Their process involved:

- Detecting conditional statements in natural language requirements using BERT model.
- Extracting these conditional in detailed, fine-grained form.
- Mapping them to a Cause-Effect-Graph, which was then used to derive the minimal set of required test cases.

The outcome of their approach was a set of acceptance test cases as decision table in natural language. Their approach served as a foundation for our Condition Scenario Generator module idea to ensure comprehensive and efficient testing.

Similar to them, we also try to detect and extract conditional statement for distinct processing, creating test case unique for testing these mentioned conditions. Unlike them, we use GPT-4 for this process.

This component receives input as a JSON object containing a list of interactive elements and their associated conditions. This input is provided by the Condition Extractor module and is already filtered to focus only on text fields. An example input of module Condition Scenario Generator is illustrated in Figure 3.6

Process Upon receiving the input, this component processes in the following steps:

1. **Generating Scenarios:** For each invalid condition, the module generates a test scenario. The generated scenario names are clear and organized to facilitate subsequent steps.

```

1 {
2   "Username": {
3     "condition": {
4       "valid": "must be over 8 characters and below 30 characters, must be entered
5     ",
6       "invalid": "below 8 characters or over 30 characters, has been registered"
7     },
8     "type": "text field"
9   },
10  "Password": {
11    "condition": {
12      "valid": "must be over 8 characters and below 30 characters, contains at
13      least one of each kind: a normal character, a capitalize character, a number and a
14      special character, cannot be the same as the username",
15      "invalid": "below 8 characters or over 30 characters, does not contain at
16      least one of each kind: a normal character, a capitalize character, a number and a
17      special character, is the same as the username"
18    },
19    "type": "text field"
20  },
21  "Email": {
22    "condition": {
23      "valid": "must be an exist email with the right format",
24      "invalid": "an inexistent email or email with the wrong format"
25    },
26    "type": "text field"
27  }
28 }

```

Figure 3.6: Example input of the Condition Scenario Generator agent

2. Ensuring Relevance: The module strictly follows the rule to avoid generating scenarios for elements or conditions not specified in the input, ensuring the relevance and accuracy of the scenarios.

This process is guided by the following rules, which were created to solve the problems mentioned later:

- Do not generate test scenarios for elements or conditions not mentioned in the given element list.
- Generate only the scenario names without including the detailed steps.

The output of the Condition Scenario Generator module is a list of test scenario names, formatted as strings. Take a look at the illustration of the output in Figure 3.7.

```
1. Username below 8 characters
2. Username over 30 characters
3. Username has been registered
4. Password below 8 characters
5. Password over 30 characters
6. Password does not contain at least one normal character
7. Password does not contain at least one capitalize character
8. Password does not contain at least one number
9. Password does not contain at least one special character
10. Password is the same as the username
11. Email is an inexistent email
12. Email with the wrong format
```

Figure 3.7: Example output of the Condition Scenario Generator agent

The main purpose of this component is to solely treat the conditional statement to ensure the coverage. This decision is based on our observation when collecting SRS document, which usually describe successful cases and occasionally outline invalid cases separately or even not describe invalid cases. In most scenarios, BA document the ideal, successful flows of interactions, ensuring the primary functionalities are well-covered. The Main-Flow Scenario Generator and Sub-Flow Scenario Generator modules are dedicated to covering these valid scenarios comprehensively, but these modules sometime miss condition mentioned in the flow. By concentrating on invalid conditions, the Condition Scenario Generator ensures that all potential errors and edge cases are thoroughly tested. Additionally, generating test scenarios for valid conditions within this module would be redundant and inefficient, as it duplicates the efforts of other modules designed for that purpose.

One of the challenges faced in designing this module was ensuring the accurate generation of test scenarios for complex conditions without missing any conditions. In addition, this module often tends to create test scenarios based on its knowledge and predictions, sometimes totally different from the use case description, leading to deviations and the production of test cases that are not suitable for testing the application feature. To address this, we filtered elements from elements extracted by the Condition Extractor modules and force this component focus only on relevant

conditions inputted.

One another problem is that module generate unreasonable scenarios. In the runs time, some of the produce test scenarios involve too many information that are not high-level descriptions. The detailed test scenarios with test steps, objective and so on. Having to create too many information in one run can cause misleading results that leading to less valuable test scenarios. To solve this problem, we provide module with prompting rule to make it 'generate only the scenario names without including the detailed steps'.

The Condition Scenario Generator module plays a particular role in our approach by generating clear and organized test scenarios for use case specification's conditions. This enhances the overall quality and efficiency of the testing process, ensuring comprehensive coverage of all relevant conditions.

3.3.2 Main-Flow Scenario Generator

The Main-Flow Scenario Generator component (Block 4 from Figure 3.1), focus on applying GPT-4's potential to generate comprehensive test scenarios for the main flow of a use case. This component aims to ensure that all possible scenarios that can occur within a single use case are identified and tested.

Initially, this component is prompted to generate all possible scenarios based on the details provided in the main flow of the use case. This involves predicting various sequences of user interactions that could potentially lead to different outcomes or states of the application. However, through multiple runs, we observed common errors. Some of the typical errors encountered include:

- **Scenario Granularity:** Some of the created scenarios seem to have too many details to be considered test scenarios. They include testing steps and other information. Predicting too much information

for each scenario can make this component appear to suggest less valuable scenarios, as well as reduce the performance of the test case generation phase because they complicate creation and validation of test cases.

- **Dependent Scenarios:** This component tends to create test scenarios that validate each individual step defined in the use case. This approach often leads to an excessive number of scenarios, many of which may be redundant and costly.

To address these issues, we have established a set of rules to guide this component in generating accurate and comprehensive scenarios. These rules help mitigate the occurrence of unwanted scenarios and hallucinations by clearly defining our requirements. The rule set focuses on the following aspects:

- **Defining a Test Scenario:** Given component a test scenario definition as a high-level description of what to test in a particular functionality or feature of the software application. It is essentially a statement that outlines a specific situation or aspect to be tested.
- **Preventing the creation of test scenarios that validate individual steps in the flow:** This component is instructed not to create scenarios that only validate individual steps in the process. Instead, each scenario must be independent and include a complete sequence of events.

Take use case Purchase in the Figure 3.2 as an example. Based on the main flow information of the use case provided, expected scenarios are created by this component including: "Successful Checkout with Single Shop Products", and "Successful Checkout with Multiple Shops Products".

3.3.3 Sub-flow Scenario Generator

The Sub-Flow Scenario Generator module (Block 5 from Figure 3.1) is designed to generate test scenarios for other flows except for the main flow within a use case, such as alternative flows, exception flows, or validation flows. We call these flows as sub-flows. The primary input for this module is the description of the main flow along with one of the sub-flows. This component ensures that specific cases, often encountered as deviations or exceptions from the main flow, are thoroughly tested.

Unlike the main flow, sub flows describe specific special events that can be reached by the user while performing the main flow. These sub-flows often represent as alternative paths or exceptional conditions that need to be verified to ensure robust application behavior. By distinguishing between main and sub-flows, our approach aims to ensure all possible the interactions and scenarios might occur within a use case are covered.

Our desired of Sub-Flow Scenario Generator module is a list of scenario names that represent meaningful transitions from the main flow to the alternative or exception flow. These scenarios are critical for testing the application's behavior under various conditions.

During the initial development and testing of the Sub-Flow Scenario Generator module, we identified several common issues that needed to be addressed to ensure the accuracy and usefulness of the generated scenarios:

- Pay attention on User Intent: The component might generate multiple scenarios based on variations in user intent (e.g., accidental vs. intentional actions). Regardless of the user's intent, the testing steps, expected results of the test cases are the same. For example, whether a user accidentally or intentionally triggers a payment failure should not result in separate scenarios, as the system's response should be tested for the action regardless of intent. Creating test scenarios that focuses on user intent results in an overwhelming number of test scenarios with minimal added value. These scenarios have high pos-

sibility to create duplicating test cases with test steps and result is all the same.

- Focus on main flow and Miss sub flow Scenarios: The component might focus too much on the main flow, missing out on critical alternative or exception paths that need testing. Since main flow often has more information and be describe in more detailed because it describe the performance with very wide range of actions and sub-flow is another branch of execution actions separated by main-flow, sometimes the sub-flow is as short as a sentence describing the violating conditions and the results.
- Dependent Scenarios: Similar to module Main-Flow Scenario Generator, this module generates test scenarios to validate each individual step outlined in the use case, leading to an enormous test scenarios.

Towards solving these problems, a set of rules was established to guide this component in generating desired test scenarios. These guideline help minimize the possibility of unexpected situations and ensure thorough testing:

- Exclusion of User Intent: Scenarios should not consider user intent (e.g., accidental vs. intentional actions). This rule ensures that the generated scenarios focus on the actions and interactions themselves rather than the motivations behind them. This helps in streamlining the scenario generation process and avoids unnecessary complexity and redundancy.
- Focus on Sub-Flows only: Scenarios should not be generated for the main flow but should test the transitions from main flow to sub-flows. This ensures that this component covers all possible deviations from the main path, including alternative and exception flows. By explicitly focusing on transitions, this component ensures that all potential

paths a user might take through the application are tested, providing a comprehensive understanding of how the system handles unexpected conditions or alternative processes. This helps in identifying edge cases that might not be created if only the main flow is tested.

- Preventing the creation of test scenarios that validate individual steps in the flow: Similar to module Main-Flow Scenario Generator guideline, this component is force not to generate scenario that only test single steps in the flow. Instead, each scenario should be a standalone sequence, avoiding partial flows and scenarios must represent complete sequences of user interactions rather than isolated steps.

For instance, in Example 3.3 if the main flow involves a user register successfully by username and the sub-flow involves registering with an existed username failures, this component would generate scenarios like "Register Failure with an Existed Username" to cover this specific deviation.

3.3.4 Scenario Refiner

The main purpose of Scenario Refiner module is to reduce redundant and duplicate test scenarios generated by the previous three scenario-generating modules (refer to Condition Scenario Generator, Main-Flow Scenario Generator, Sub-Flow Scenario Generator modules).

Redundant scenarios are often caused by hallucinations or missing information from other flows. Since the Main-Flow Scenario Generator and Sub-Flow Scenario Generator only receive and generate scenarios for one specific flow and are not aware of other flows. This is to ensure each component only handles a small piece of information each time, so that component won't raise errors of context overload, ambiguity, and performance degradation. Although splitting the use case to generate test cases is possible to decrease the error-raising ability, it may results in duplication when each component independently processes a specific flow.

Additionally, redundant scenarios could be stated as test scenarios that are hard to perform. For instance, test scenarios involving Page or System Load Failure, System Error, Network Error, or interactions with non-existent objects,... that seem to be hard to explain, present in ambiguous test steps of the test cases.

Another challenge was ensuring that all possible scenarios were covered without redundancy. At first, when we did not provide component with the use case specification, it seems not having a sufficient knowledge to assess which scenarios is redundancy with scenarios not mentioned in the use case specification or test scenarios that testing the same thing with different expressions and word usage, which is difficult to recognize unless placed in a certain context.

After we observed the problems, we conducted some research to find solutions. There are researches shown the effectiveness of GPT in self-evaluation and refinement to produce better outputs [23]–[26]. Applying these research results, we employ GPT-4 to develop Scenario Refiner module to evaluate and refine the input list of test scenarios. Furthermore, we provide component with the use case specification. This process ensures the removal of redundant and duplication scenarios, enhances the overall effectiveness of test scenario result set.

The inputs for this component involves:

- List of generated test scenarios: All the test scenarios generated by the three scenario-generating components.
- Use case specification details: The complete use case specification document.

Once the module is activated and receives the input data, it employs GPT-4 to refine the raw list of test scenarios according to a set of rules we designed:

- Assess sufficiency: Making assessment of whether the provided test

scenarios are sufficiently cover the use case according to the details of the use case specification.

- Remove redundant test scenarios: We define the redundant scenarios as the scenarios that test the features, actions, and behaviors that are not mentioned in the use case specification or the test scenarios to test the activities that hard to present, perform in real life.
- Eliminate duplicate scenarios: Agent should detect and erase all the duplicate scenarios to ensure that each scenario is unique.

The output of this component should be a refined and optimized set of test scenarios that:

- Each scenario is unique, with different objectives to result in different test cases.
- Each scenario is relevant to the use case, excluding any irrelevant scenarios.

By employing the Scenario Refiner module, we ensure a more effective and efficient set of test scenarios than directly using the results of the above test case generation modules, thereby improving the overall quality of manual test case generation from use case specifications.

3.3.5 Reason for using multiple modules instead of one module

The reason why we separate many prompts for the work of test scenario designing instead of only using a single prompt is explained as follows:

- Prevent information overload: In the software development industry, each company often has their own use case specification template with different components, and the use case specification can be full

of information, including describing multiple flows and having components to make use case specification clearer but do not contribute in the test case generation such as goals, assumptions, references, actors, authors, stakeholders, interests, and more. Inputting the use case that wasn't processed and had lots of minor details can increase the ratio of hallucination or information ignorance, which leads to producing results that do not cover all the paths.

- Apply appropriate guide: Besides the main flow, which describes the primary flow of actions to perform the use case, we have other flows that could describe the variations or alternative flow of steps that still achieve the use case success or exception flows, which describe unexpected conditions or actions that encounter errors. Each flow-described component of use case can have the flow transfer explanation that is different from others. Treating all these components the same can lead to inappropriate guidance and make guidance conflicts with each other. Divided single module into multiple modules can help treat each flow-described component with suitable rules and instructions.
- Generate specific scenarios: For use cases that have specific condition descriptions, as in the use case example in Figure 3.3 where the condition is clear and diverse: "Over 8 characters and below 30 characters and contain at least one of each kind: a normal character, a capitalized character, a number, and a special character and cannot be the same as the username." One prompt may only generate a non-specific condition by using the word "invalid," as in Figure 3.8. This non-specific test scenario generated in the test design phase will put a heavy pressure on the next phase to define and generate test cases to test all the cases of invalid conditions, proving to be ineffective in test design. And in most of the cases, the generation phase generates non-specific test cases due to the unclear test scenario's condition

that leading to miss important test cases, as exemplified in Figure 3.9.

```
1 1. Successful Registration with Valid Username and Password
2 2. Successful Registration with Valid Email
3 3. Registration with Invalid Username
4 4. Registration with Invalid Password
5 5. Registration with Existed Username
6 6. Registration with Invalid Email
```

Figure 3.8: Example Test Scenario Generated by Single Prompt

```
1 Test Case: Test Case 1
2 testCaseName: Register with a password missing a special character
3 objective: Verify that the system shows a notification that the password is invalid when
   the password is missing a special character
4 testSteps:
5   - Step 1: Navigate to the registration page.
6   - Step 2: Fill in the username field with a valid username that is over 8 characters
   and below 30 characters.
7   - Step 3: Fill in the password field with a password that is over 8 characters, below
   30 characters, contains at least one normal character, one capitalize character, and
   one number but no special character.
8   - Step 4: Press the 'Register' button.
9 expectedResult: System shows a notification that the password is invalid.
10
11 Test Case: Test Case 2
12 testCaseName: Register with a password shorter than 8 characters
13 objective: Verify that the system shows a notification that the password is invalid when
   the password is shorter than 8 characters
14 testSteps:
15   - Step 1: Navigate to the registration page.
16   - Step 2: Fill in the username field with a valid username that is over 8 characters
   and below 30 characters.
17   - Step 3: Fill in the password field with a password that is shorter than 8 characters
   but includes at least one normal character, one capitalize character, one number, and
   one special character.
18   - Step 4: Press the 'Register' button.
19 expectedResult: System shows a notification that the password is invalid.
20
21 Test Case: Test Case 3
22 testCaseName: Register with a password identical to the username
23 objective: Verify that the system shows a notification that the password is invalid when
   the password is identical to the username
24 testSteps:
25   - Step 1: Navigate to the registration page.
26   - Step 2: Fill in the username field with a valid username that is over 8 characters
   and below 30 characters.
27   - Step 3: Fill in the password field with a password that is identical to the username
   but meets all other criteria for a valid password.
28   - Step 4: Press the 'Register' button.
29 expectedResult: System shows a notification that the password is invalid.
```

Figure 3.9: Example Test Cases Generated by Non-specific Test Scenario

All these comparisons will be demonstrated in Section 4 where our method is compared with the Baseline method using only 1 module for each phase, specifically in the section 4.3.

3.4 Test Case Generation

3.4.1 Test case Generator

The primary purpose of the Test Case Generator module (Block 7 from Figure 3.1) is to generate test cases based on scenarios produced from Test Design Phase and the comprehensive details of the use case specification. Using GPT-4, this component ensures the each test scenario is used to generate actionable and precise test cases, that are essential for effective software testing.

The inputs for this component includes:

- Test scenario: These are test scenario set results after being refined by Scenario Refiner module. Each test scenarios from that list will be used to generate corresponding test cases to test only that scenario.
- Use case specification details: The complete use case specification document.

The main reason we do not put in the full list of test scenario set but only one scenario for each run time is to avoid task misunderstandings by the module used to generate test cases. Inputting the entire set increases the likelihood of generating only one test case per scenario in every run time that could decrease the coverage of generated test cases and increase the rate of generating test case for multiple cases and connect them by 'or'.

An example of this component's input of scenario Password Same as Username of use case Registry is shown in the figure 3.10.

During implementing, we encountered several challenges that reduced the efficiency of the generating test cases process. Some of the typical errors encountered include:

- Mismatched Scenario Names: Generated test cases that do not align directly with scenario names, causing test cases to be either too broad or not sufficiently targeted for effective testing.

```

1 Test scenario: Password Same as Username
2 Use case specification:
3 Use case name: Registry
4 Actor: Learner
5 Precondition: Learner is in Registry page.
6 Main flow:
7     Step 1: Learner fills in the username field by a valid username, a valid username
      must be over 8 characters and below 30 characters.
8     Step 2: Learner fills in the password field by a valid password, a valid password
      must be over 8 characters and below 30 characters, contains at least one of each kind
      : a normal character, a capitalize character, a number and a special character. A
      valid password cannot be the same as the username.
9     Step 3: Learner press "Registry" button
10    Step 4: System redirects learner to Home page
11 Alternative flow 1: Learner register by email
12     At step 1 of the basic flow: Learner clicks on the Mail icon
13     Step 2: Learner fills in the email field by a valid email, a valid email must be an
      exist email with the right format
14     Go back to step 2 in the basic flow and continue with the steps from step 2.
15 Exception flow 1: Learner enters invalid username
16     At step 1 of the basic flow: Learner fills in the username field by an invalid
      username
17     Go back to step 2 in the basic flow and continue with the steps from step 2.
18     At step 4 of the basic flow: System shows a notification that the username is invalid
19 Exception flow 2: Learner enters invalid password
20     At step 2 of the basic flow: Learner fills in the password field by an invalid
      password
21     Go back to step 3 in the basic flow and take only step 3.
22     At step 4 of the basic flow: System shows a notification that the password is invalid
23 Exception flow 3: Learner enters an existed username
24     At step 1 of the basic flow: Learner fills in the username field by an username that
      has been registered
25     Go back to step 2 in the basic flow and continue with the steps from step 2.
26     At step 4 of the basic flow: System shows a notification that this username has been
      registered and ask the learner to choose another username.
27 Exception flow 4: Learner enters invalid email
28     At step 2 of the alternative flow 1: Learner fills in the email field by an invalid
      email, an invalid email must be an inexistent email or email with the wrong format
29     Go back to step 3 in the alternative flow 1 and continue with the steps from step 3.
30     At step 4 of the basic flow: System shows a notification that the mail is invalid
31 Postcondition: Regist successfully

```

Figure 3.10: Example input of the Test Case Generator agent

- **Redundant Test Cases:** This component tended to generate multiple test cases that test many aspects of use cases despite the content of test scenario. This led to redundancy, where testers had to sift through excessive tests that essentially validated the same functionality.
- **Ambiguous Test Steps:** Test steps generated by this component are often unclear and lacked independence. Testers frequently needed to reference other test cases or use case specifications to understand the context and execute steps correctly. Examples of ambiguous test

steps: "Starting from step 1 to step 4 in use cases", "Similar to step 1 to 3 mentioned in Test Cases 1". This ambiguity slowed down the testing process and increased the likelihood of errors.

- **Inconsistent Structure and Missing Information Test Cases:** Test cases generated have varied formats and sometimes key details like objectives, specific steps, expected results are omitted. This inconsistency can make it difficult for testers to understand and follow the test cases. A lack of uniformity in the test case format can also complicate the process of reviewing and validating the test cases, leading to potential errors and inefficiencies.

To address these issues, we implement rules to force this component generate test cases with:

- **Clear Steps:** Ensures that each test step is self-explanatory and independent. This rule is used to make sure every test step is described clearly and comprehensively, so that testers can execute the steps without needing to refer to other test cases or external documentation. By having well-defined steps, test cases generated could maintain the independence and clarity that reducing potential confusion and errors during testing.
- **No References:** Eliminates dependency between test cases. This rule prohibits referencing other test cases or instructions like "do as mentioned." This ensures that each test case is self-contained, meaning testers do not have to cross-reference or rely on information from other test cases. This independence is vital for efficient test execution and prevents the propagation of errors that could occur if a referenced test case changes. It also makes each test case easier to understand and manage, as all necessary information is contained within the test case itself.

- Align Actions to Match Scenario: Guarantees that the test case objectives and actions are aligned with the scenario name. This alignment is used for clarity and relevance, ensuring that the test case directly addresses the scenario described.

```
1 {  
2   "Test Case": {  
3     "testCaseName": "Clear name of the test case so tester know what to test when they  
4       first read",  
5     "objective": "Verify who doing what action or function in the test case and the  
6       summary of the final result of the test case",  
7     "testSteps": [  
8       "Step 1: Describe the step.",  
9       "Step 2: Describe the step.",  
10      "Step 3: Describe the step."  
11    ],  
12    "expectedResult": "You inform the tester what should they see after doing all the  
    steps"  
13  }  
14 }
```

Figure 3.11: Test case format in prompt

To solve the problem related to creating Inconsistent Structure and Missing Information test cases, we define our desired output of Test Case Generator as a structured set of test cases in structured JSON format, each test case must include a test case name, objectives, testing steps, expected results. Clearly defining the desired output make it easier to modules with a satisfactory output template, ensuring clarity and consistency in all outputs. We design tempalte of output as illustrate in Figure 3.11. By providing this template, this component's response results follow the template and do not generate missing or redundant sections.

From the improvement efforts mentioned above, the Test Case Generator modules has shown a better results of test cases generated: test cases followed template and mentioned all necessary sections in consistency structure. An example of test cases generated module output of Scenario Password Same as Username is shown in the Figure 3.12.

However, despite the improvements achieved through the implementation of the above rules and the structured JSON format for test case

```

1 {
2   "Test Case 1": {
3     "testCaseName": "Password Same as Username",
4     "objective": "Verify that the system prevents the user from registering when the
5       password is the same as the username.",
6     "testSteps": [
7       "Step 1: Navigate to the registration page.",
8       "Step 2: Enter a valid username in the username field (ensure the username is over
9       8 characters and below 30 characters).",
10      "Step 3: Enter the same value as the username in the password field, ensuring it
11      meets the password criteria (over 8 characters and below 30 characters, contains at
12      least one normal character, one capitalize character, one number, and one special
13      character).",
14      "Step 4: Click on the 'Registry' button."
15    ],
16    "expectedResult": "The system should display a notification that the password is
17      invalid because it cannot be the same as the username.",
18    "explanation": "This test case directly addresses the scenario 'Password Same as
19      Username' by attempting to register with a password identical to the username, which
20      is against the use case specifications."
21  },
22  "Test Case 2": {
23    "testCaseName": "Valid Registration with Different Username and Password",
24    "objective": "Verify that the system allows the user to register when the password
25      and username are different and both meet their respective criteria.",
26    "testSteps": [
27      "Step 1: Navigate to the registration page.",
28      "Step 2: Enter a valid username in the username field (ensure the username is over
29      8 characters and below 30 characters).",
30      "Step 3: Enter a valid password in the password field that is not the same as the
31      username (ensure the password is over 8 characters and below 30 characters, contains
32      at least one normal character, one capitalize character, one number, and one special
33      character).",
34      "Step 4: Click on the 'Registry' button."
35    ],
36    "expectedResult": "The system should redirect the user to the Home page, indicating a
37      successful registration.",
38    "explanation": "This test case verifies the system's functionality in allowing
39      registrations that comply with the criteria for usernames and passwords, specifically
40      ensuring that the password is not the same as the username."
41  }
42 }

```

Figure 3.12: Example output of the Test Case Generator agent for Scenario Password Same as Username

generation, challenge remains: redundancy test cases generated that misaligned with the given test scenarios. To mitigate this, we designed the Test Case Validator module (Block 8 in Figure 3.1) to review and provide feedback on whether the generated test cases accurately reflect the given test scenario. This additional step helps refine and validate the test cases, enhancing their quality and relevance.

3.4.2 Test Case Validator

```
1 Test scenario: Password Same as Username
2 {
3   "Test Case 1": {
4     "testCaseName": "Password Same as Username",
5     "objective": "Verify that the system prevents the user from registering when the
6       password is the same as the username.",
7     "testSteps": [
8       "Step 1: Navigate to the registration page.",
9       "Step 2: Enter a valid username in the username field (ensure the username is over
10        8 characters and below 30 characters).",
11       "Step 3: Enter the same value as the username in the password field, ensuring it
12        meets the password criteria (over 8 characters and below 30 characters, contains at
13        least one normal character, one capitalize character, one number, and one special
14        character).",
15       "Step 4: Click on the 'Registry' button."
16     ],
17     "expectedResult": "The system should display a notification that the password is
18       invalid because it cannot be the same as the username.",
19     "explanation": "This test case directly addresses the scenario 'Password Same as
20       Username' by attempting to register with a password identical to the username, which
21       is against the use case specifications."
22   },
23   "Test Case 2": {
24     "testCaseName": "Valid Registration with Different Username and Password",
25     "objective": "Verify that the system allows the user to register when the password
26       and username are different and both meet their respective criteria.",
27     "testSteps": [
28       "Step 1: Navigate to the registration page.",
29       "Step 2: Enter a valid username in the username field (ensure the username is over
30        8 characters and below 30 characters).",
31       "Step 3: Enter a valid password in the password field that is not the same as the
32        username (ensure the password is over 8 characters and below 30 characters, contains
33        at least one normal character, one capitalize character, one number, and one special
34        character).",
35       "Step 4: Click on the 'Registry' button."
36     ],
37     "expectedResult": "The system should redirect the user to the Home page, indicating a
38       successful registration.",
39     "explanation": "This test case verifies the system's functionality in allowing
40       registrations that comply with the criteria for usernames and passwords, specifically
41       ensuring that the password is not the same as the username."
42   }
43 }
```

Figure 3.13: Example input of the Test Case Validator agent for Scenario Password Same as Username

The Test Case Validator module (Block 8 in Figure 3.1) is specifically designed to validate the test cases generated in the previous steps (Test Case Generator module results). Since in the Test Case Generator creates test cases for each scenario, we witnessed issues as redundant and misaligned test cases can arise. This problem needs to be resolved to ensure that the generated test cases are accurate, relevant, and effective for the

given test scenarios. For that reason, we designed this component.

```
1 {
2   "Test Case":{
3     "testCaseName": "Clear name of the test case so tester know what to test when they
4       first read",
5     "objective": "Verify who doing what action or function in the test case and the
6       summary of the final result of the test case",
7     "testSteps": [
8       "Step 1: Describe the step.",
9       "Step 2: Describe the step.",
10      "Step 3: Describe the step."
11    ],
12    "expectedResult": "You inform the tester what should they see after doing all the
13      steps",
14    "reflect": "yes/no",
15    "explanation": "explain why you think this test case reflect the given test scenario
16      or not",
17  },
18 }
```

Figure 3.14: Test case validated format in prompt

After the Test Case Generator creates test cases for each scenario, the Test Case Validator module reviews these test cases to confirm they adequately test the given scenario. This validation step is crucial for filtering out any irrelevant or inaccurate test cases, ensuring that only the most pertinent ones are included in the final result set. The component's input is illustrated in the Figure 3.13, including:

- Test Scenario: The given Test Scenario was used to create set of test cases by the Test Case Generator module.
- Test Cases: A set of test cases produced by the Test Case Generator module, intended to test the given scenario.

The output of the Test Case Validator module is a list of cases after reviewing, formatted as JSON. In order to ensure consistency of output results, no loss of test case sections and create convenience in extracting information sections of each test case, we provide the component with a template for component's response. The provided template has the same structure as the module Test case generator template but with the addition of "reflect" and "explanation" sections show in Figure 3.14. The value of

```

1 {
2   "Test Case 1": {
3     "testCaseName": "Password Same as Username",
4     "objective": "Verify that the system prevents the user from registering when the
5       password is the same as the username.",
6     "testSteps": [
7       "Step 1: Navigate to the registration page.",
8       "Step 2: Enter a valid username in the username field (ensure the username is over
9         8 characters and below 30 characters).",
10      "Step 3: Enter the same value as the username in the password field, ensuring it
11        meets the password criteria (over 8 characters and below 30 characters, contains at
12        least one normal character, one capitalize character, one number, and one special
13        character).",
14      "Step 4: Click on the 'Registry' button."
15    ],
16    "expectedResult": "The system should display a notification that the password is
17      invalid because it cannot be the same as the username.",
18    "reflect": "yes",
19    "explanation": "This test case directly targets the test scenario 'Password Same as
20      Username' by attempting to perform an action that should be prevented by the system
21      according to the scenario's requirements. It checks if the system correctly
22      identifies and blocks an attempt to use a password identical to the username, which
23      is the core of the given test scenario."
24  },
25  "Test Case 2": {
26    "testCaseName": "Valid Registration with Different Username and Password",
27    "objective": "Verify that the system allows the user to register when the password
28      and username are different and both meet their respective criteria.",
29    "testSteps": [
30      "Step 1: Navigate to the registration page.",
31      "Step 2: Enter a valid username in the username field (ensure the username is over
32        8 characters and below 30 characters).",
33      "Step 3: Enter a valid password in the password field that is not the same as the
34        username (ensure the password is over 8 characters and below 30 characters, contains
35        at least one normal character, one capitalize character, one number, and one special
36        character).",
37      "Step 4: Click on the 'Registry' button."
38    ],
39    "expectedResult": "The system should redirect the user to the Home page, indicating a
40      successful registration.",
41    "reflect": "no",
42    "explanation": "Although this test case is related to the registration process and
43      involves the relationship between the username and password, it does not directly
44      test the scenario 'Password Same as Username.' Instead, it tests a positive scenario
45      where the username and password are different and meet the criteria, which is a
46      different flow from the one specified in the given test scenario."
47  }
48 }

```

Figure 3.15: Example output of the Test Case Validator agent

section "reflect" is "yes" or "no" indicating whether the test case accurately reflects the given test scenario. While the value of section "explanation" is A detailed justification for the validation decision, explaining why the test case reflects or does not reflect the scenario.

To review the reflect of test cases, The Test Case Validator follows a process of:

- **Input Reception:** The component receives the test scenario and the corresponding test cases generated by the Test Case Generator.
- **Criteria Checking:** The component evaluates each test case against the predefined validation criteria (test case name, objective, test steps, and expected output).
- **Reflection Assessment:** The component marks each test case as either reflecting or not reflecting the scenario based on the evaluation.
- **Explanation Generation:** The component provides a detailed explanation for each decision, justifying why a test case was marked as reflecting or not reflecting the scenario.

By strictly following validation criteria, the component ensures that only correct and relevant test cases are determined to reflect as "yes" in the result set. The output example is illustrated in the Figure 3.15. And to get the best results for the final set, we use code to filter and get all test cases whose reflect value is yes and present in text form.

The Test Case Validator module is a pivotal component in our approach. By rigorously validating each test case against specific criteria, it ensures the accuracy, relevance, and effectiveness of the test cases, contributing to a more efficient and reliable testing process.

Chapter 4

EXPERIMENTS AND RESULTS

4.1 Experimental design

4.1.1 Dataset

To conduct our experiment, we collected use case specifications written by students or publicly available on GitHub. The only requirement for the data was that the events needed to be described in separate flows.

We assembled a data set consisting of four real-world applications, including 44 use cases. Each application has different formats and writing styles for use cases. These applications are briefly described as follows:

1. **English Learning Website:** A web-based application that includes 12 use cases, each describing flows and test conditions in a very detailed manner. The use cases are comprehensive, covering various aspects of the application’s functionality, such as user registration, course enrollment, lesson progression, and quiz taking.
2. **Book Catalog Website:** This application ¹ is written in a straightforward format and includes 15 use cases. Each use case has at most 2 flows and incorporates many conditions, presented in a simple manner. The use cases described in this application focus mainly

¹<https://github.com/Gr0ki/book-catalog>

on managing and interacting with three objects: Book, Author and Genre.

3. **Online Shopping Platform:** This application contains 5 use cases, each describing flows and test conditions. Some of the use cases are related to other use cases. Only important features are mentioned, and the only actor is the customer.
4. **Hotel Management System:** An application ² with 12 use cases, each detailing only the flows. The conditions are mostly unclear, and the results for events are also ambiguous. The use cases typically address functionalities such as room booking, check-in/check-out processes, managing reservations, and customer service interactions.

Refining the data

The raw dataset is documents in form of PDF or Word files. Since we did not have any process to refine these files into separate use cases, we manually extract each use case into independent text files. This involved carefully reading through each document, identifying distinct use cases, and saving them as individual text files for easier analysis and processing. This manual extraction ensured that each use case was accurately separated and could be worked with independently in subsequent steps of our experiment.

For some use cases, we found the descriptions difficult to understand, even for humans. In such cases, we rewrote these use cases to ensure clarity. This step was necessary because we cannot expect our method to generate accurate test cases based on descriptions that we, ourselves, cannot understand. If we do not understand the original text, it becomes impossible to verify the correctness of our method's responses. Therefore, rewriting these unclear use cases allowed us to ensure that both the input to method and the subsequent test cases it generates are understandable

²<https://github.com/aliasar1/Hotel-Management-System-Documentation.git>

and verifiable.

4.1.2 Evaluation metrics

We introduce metrics for evaluating results generated by our method, including test scenarios generated by the first phase and test cases generated by the second phase, which is also our final result.

Test Scenario's Evaluation Metrics

Correct: Scenarios that represent detailed actions mentioned in the use case, each being unique, logical and crucial for covering the use case. Removing these scenarios could lead to a decrease in flow coverage. Additionally, we also classify scenarios that our method generates independently, without clear guidance from the use case specifications. If these independently generated scenarios is rated could be tested by human, we also count them as valid scenarios. These scenarios are expected to generate testable test cases.

Duplicate: Scenarios that test the same thing as one of the scenarios that appeared before, or it has the same meanings but different wordings.

Incorrect: scenarios that cannot be tested due to illogical or it's detail is not clearly specified in the use case. These scenarios have high chance making our method to generate incorrect test steps, leading to misunderstandings or making it difficult for testers to follow the test steps, creating test case that cannot be executed. This mostly occurs because our method relies on its own assumptions to make decisions about the contents of these test cases in the absence of specific information. As a result, we consider these scenarios as invalidity.

Test Case's Evaluation Metrics

Correct: Test cases with testable test steps, correct expected output and reflected to the scenario and use case they represent for. These test

cases should be comprehensive, allowing testers to understand and execute them. Each step and expected outcome should be clearly stated to ensure the test case can be followed thoroughly and effectively validated against the specified requirements.

Duplicate: Test cases with the same test steps and same expected output with test case that have appeared before. Test cases in this category must also testable test cases, same title or same expected result does not count if it cannot be executed. Test cases with same title but has different test steps, or test cases with same test steps but different test data is not consider as duplication.

Incorrect: Test cases with ambiguous test steps or wrong expected output. The steps described in these test cases may be difficult to follow, illogical, or impossible to execute. Figure 4.1 illustrates one case of invalidity with a test case has its test steps refer to another test case.

```
Test Case: Test Case 2
testCaseName: Verify Seller Contact Functionality
objective: Verify that users can initiate contact with the seller through the provided
           contact option on the product information page.
testSteps:
  - Step 1: Follow steps 1 and 2 from Test Case 1 to view a product's information.
  - Step 2: Click on the 'Contact Seller' option.
  - Step 3: Observe the system's response.
expectedResult: The system should navigate the user to the platform's messaging system to
                 allow direct contact with the seller.
reflect: yes
explanation: This test case is a direct application of the 'Seller Contact Initiated'
            part of the test scenario, ensuring that the functionality for initiating contact
            with the seller is working as intended.
```

Figure 4.1: Example of invalid test case with ambiguous test steps

Coverage

Coverage is also an important aspect when consider the main purpose of our work is to assist software testing process. Since our approach is based on Black Box Testing, relying solely on the use case specifications, code coverage is not applicable for us. We also do not expect our method to create tests for cases which are not described in the use case, this kind

of test could be achieve by human by doing exploration testing, and we acknowledge this as one point our method could not match with human performance.

Our coverage metric is derived from the information provided in the use case specification, specifically focusing on how much of that information the generated test cases manage to cover. Our only requirement for the input is to follow any format that describes flows of events.

$$\text{Coverage} = \frac{CG}{CF} \quad (4.1)$$

Number of cases found (CF): We count each flow as an important case that needs to be covered. However, this is not always sufficient. Users might use a format that includes additional important information, such as success or failure conditions declared within a single flow, indicating that a flow could have multiple scenarios. We also count each of these as separate scenarios. It should be noted that this counting is done manually, which means there is a possibility of errors or omissions. In short, this is maximum cases could happen in an application described in use case.

Number of cases generated (CG): We verify each cases identified by our method to determine if there is a corresponding test case. We then assess whether that test case is testable or untestable. If a testable test case exists for a scenario, we consider that scenario to be covered.

4.1.3 Baseline method

To evaluate the effectiveness of our methodology, we implemented a baseline method for comparison. This method is based on previous research on this topic to establish a basic route for deriving test cases from use cases.

It divides into two phases: the test design phase for generating test scenarios and the test case generation phase for generating test cases from use case specifications. This method also uses Zero-shot prompting techniques and is provided human knowledge on testing topic to instruct the

generation process. All given human knowledge is similar with those in use in our method.

Basically, it simulates how people interact with ChatGPT for getting test cases generated, using only one prompt for each phase and remove the refining phases. We also run this method with the same set of data mentioned before. Figure 4.2 clearly illustrates the baseline method.

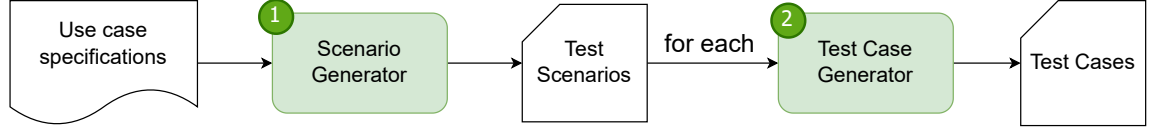


Figure 4.2: Baseline method

The main purpose we doing this experiment is to prove the necessary of dividing task into smaller pieces as we use more prompts, and also the importance of the refining phase.

4.1.4 Procedure

Experiment without human involvement

In this experiment, we only defined the use cases and then awaited the generation of a test case set, which was used to thoroughly test the application.

We then manually rank each test case in the result set into groups define in Section 4.1.2 by reading all test cases generated and check for essential detail. We do not rate the test scenarios as it is solely the result of phase one and serves as input for phase two, we still give out these test scenarios in excel files for more context to judge if the test case reflect the corresponding test scenarios or not. After that, we calculate the percentage of each category to achieve the chance of appearance of each kind of test cases.

Finally, we calculate the coverage of the test case by checking if each test case belongs to the Correct group could cover for any important detail described in the use case specification.

Experiment with human involvement

For this experiment, we ourselves interceded in the generation process of the method, by removing the redundant result of the phase one. To conduct this experiment, we interrupt the flow of the method after the phase one - the test design phase has finished it works. Then, we manually classified these scenarios into defined groups mentioned in Section 4.1.2. Only test scenarios belonged to the Correct group is used as input for the next phase. We also calculate the percentage of each category to acknowledge the error rate of this phase. By doing this, we could also identify if the result from this phase heavily affect the next phase.

We then manually rank each test case, our method's final result, into groups define in Section 4.1.2 by reading all test cases generated and check for essential detail. Finally, we calculate the percentage of each category to achieve the chance of appearance of each kind of test cases.

Comparing with baseline method

We conducted this experiment by running independently two methods. To ensure the fairness, while building this baseline method, we inject the knowledge about testing same with the knowledge we provide our method's module.

We pause our method after phase one to evaluate the test design phase, as the baseline's test design phase also differs from our method. Then, we selected only the scenarios belongs to the Correct category (mentioned in Section 4.1.2) to carry on the next phase.

After finished running the test case generation phase, we evaluated the test cases of both method by categorized them into group mentioned in

Section 4.1.2.

Evaluation method

We will independently conduct a manual evaluation of the result set based on the information provided in the use case specifications. Upon completing our individual evaluations, we will compare our results. In cases where discrepancies arise between our evaluations, we will engage in discussions to reach a consensus and unify our final evaluation.

4.2 Result

We evaluate our method based on many aspects. First, we assess the overall performance, then we evaluate independently for each phase of the method (our method has two phase, as shown in Figure 3.1) to determine method effectiveness on generating manual test cases from use cases. By analyzing each phase separately, we can identify the strengths and weaknesses of our approach and better understand the contribution of each phase to the overall effectiveness of the method.

4.2.1 Evaluation on Overall

A. Evaluation on quantity and accuracy

System	Use Case	Test Scenario	Test Case
English Learning	12	74	101
Online Shopping	5	69	102
Book Catalog	15	102	140
Hotel Management	12	63	84

Table 4.1: Total test scenarios and test cases generated

Table 4.1 represents our running result. The **Use case** column is the amount of use cases act as input for each system. The **Test Scenario**

column is the number of test scenarios generated by our propose based on those use case specification. The **Test Case** column is the amount of test cases generated by our propose for each system.

As shown in Table 4.1, difference format and difference in how detail the use case could affect the overall results. The Online Shopping System, despite having only five use cases, has nearly the same number of scenarios as the Hotel Management System, which has twelve use cases. This is due to the complexity and numerous flows in the Online Shopping System’s use cases. Conversely, the Hotel Management System’s use cases are relatively simple, resulting in fewer scenarios and test cases in the final phase. This demonstrates how the structure and detail of use cases impact the number of generated scenarios and test cases.

Besides the number of test cases generated, we also consider if the test case is usable or not. Below is our table featuring the accuracy of test cases by our propose.

System	Correct(%)	Duplicate(%)	Incorrect(%)
English Learning	88.12	9.90	1.98
Online Shopping	63.73	32.35	3.92
Book Catalog	76.43	12.86	10.71
Hotel Management	64.29	22.62	13.09

Table 4.2: Accuracy of test case generated by method

Table 4.2 presents three categories we divided each test case into from the final test case set generated by our propose with each column is a metrics defined in Section ??.

Duplicate and incorrect categories have test cases that we considered redundant test cases since they create more work for human to remove them. For these test cases, we do not need them or we cannot use them, the only thing we could do is remove them from the set. We observed that duplication test cases mainly driven from the duplication test scenario’s result comes from test design phase. While incorrect test cases are often a consequence of hallucinations.

According to the table, we come to a conclusion that complexity of the use case’s specification could affects the final results. Systems with flow divided in easy to follow format and clearly defined conditions, like the English Learning Website, often achieves higher accuracy and fewer redundant test cases. In contrast, system with simpler format and detail such as the Hotel Management System faces higher rates of duplicate and incorrect test cases.

B. Evaluation on coverage

System	CF	CC	Coverage(%)
English Learning	55	53	96.36
Online Shopping	43	43	100
Book Catalog	62	56	90.32
Hotel Management	36	32	88.89

Table 4.3: Coverage evaluation

There are scenarios which ours method could not possibly generate valid test scenario or valid test case for it. This usually occurs when ambiguous information is described, making our method to use it assumption for generating test cases. For example, error events might be mentioned without clearly stating the reasons that could potentially cause the error. One other reason that makes our approach to fail from reaching full coverage is the poor performance of refining agents, which leads to accidentally removing important test scenarios in test design phase.

The Online Shopping System in the final row is written in a very detail manner, it also has the highest duplicating proportion in test cases. We predict that maybe the high amount of test cases generated not only increase the duplicated test cases, but also somehow increase the coverage as the quantity is huge compare to the number of use case specifications inputted.

To gain a clearer understanding of the issues with the method, we

conducted experiments to separately evaluate the test design phase and the test case generation phase. This allowed us to identify which phase is primarily responsible for the generation of unusable test cases and the limitations of each phases.

4.2.2 Evaluation on test design phase

System	Correct(%)	Duplicate(%)	Incorrect(%)
English Learning	98.65	0	1.35
Online Shopping	92.75	7.25	0
Book Catalog	97.06	0.98	1.96
Hotel Management	87.31	9.52	3.17

Table 4.4: Accuracy of test scenario generated in test design phase

For each scenario generated, we investigate if it belongs to one of three scenario’s evaluation metrics stated in Section ??.

As shown in table 4.4, the duplication percentage of scenarios generated in this phase is proportional to the duplication percentage of test cases in Table 4.2. By contrast, The incorrect scenarios do not show any strong effect on the final result, since there are case in which the invalid proportion does not change much like English Learning Website, or case that invalid proportion rapidly rises like Book Catalog Website. This indicates that while this phase generates invalid scenarios, invalid scenarios do not significantly impact the overall accuracy of the test cases, but duplication in this phase causes heavy consequences as we carry on this result for generating test cases.

The main problem shown in this phase is the duplication in test scenario. Although we have addressed an agent for removing duplicated test scenario, the Scenario Refiner (Block 6 from 3.1) could not remove all duplication due to GPT-4’s limited abilities. Below is a sample result to demonstrate the problem in this phase.

-
1. Successful Deletion Scenario
 2. Deletion Cancellation Scenario
 3. User Cancels Deletion
 4. User Navigates Away Using Menu Before Confirming Deletion
-

Figure 4.3: Example of duplication still remains after refining

In the figure 4.3, scenario number two (Deletion Cancellation Scenario), scenario number three (User Cancels Deletion) and even scenario number four (User Navigates Away Using Menu Before Confirming Deletion) all refer to the same cancel action performed by the same actor in that use case. Due to differences in wording, the Scenario Refiner agent was unable to recognize that these two scenarios are identical, resulting in their failure to be filtered out as duplicates. Consequently, if we consider each scenario to generate at least one test case, we would have at least two duplicated test cases in the final result.

4.2.3 Evaluate on test case generation phase

A. Test Case Generation with Correct Test Scenario Input:

For the experiment on the effectiveness of this phase, we manually removed duplicate and incorrect scenarios from the scenario set produced by the test design phase. By doing this, we can control the input and ensure fairness, allowing us to identify the problems specific to this phase without carrying over issues from the previous phase. For example, from the result in 4.3, we remove scenario number three and four for all remaining scenarios in the set to be unique.

System	Old TS	Old TC	Input TS	New TC
English Learning	74	101	72	95
Online Shopping	69	102	52	81
Book Catalog	102	140	74	107
Hotel Management	63	84	52	70

Table 4.5: Test case generation result

The raw source of scenario for testing this phase is taken fully from the scenario set represented in table 4.4, we removed all the duplicate and incorrect scenarios from this source to form our set of human-refined scenario, which is shown in **Input TS** column in table 4.5.

System	Correct(%)	Duplicate(%)	Incorrect(%)
English Learning	90.52	2.11	7.37
Online Shopping	75.31	6.17	18.52
Book Catalog	81.31	7.48	11.21
Hotel Management	68.57	17.14	14.29

Table 4.6: Accuracy evaluation on new test case set

Duplication problem still appears as in **Duplication** column from table 4.6 after we remove duplicate scenarios. This is because of the wrongly assume of Test Case Validator agent of whether test cases generated reflect the scenario or not. Reflected test case is test case can be use for testing only the mentioned scenario, this is for limiting the appearance of duplication. An example for this case is in the below figure, these are test cases generated for scenario **Navigation to a specific page of books**, but their contents are not reflected to the mentioned scenario, causing duplication with test case generated by other pagination scenarios.

The problem lies in column **Incorrect** as shown in table 4.6, as it represents the proportion of untestable test cases comparing to total test cases generated. The probability of incorrect test cases increases as the number of test case becomes smaller when we also look at table 4.2. We find out this is due to the reduction of duplicate but testable test cases shown in **Duplicate**, but incorrect test case remains unchanged or increase before and after human-refined on input test scenario sets. We assume that the reason why there is appearance of incorrect test cases is because of agents assumption if there is ambiguous information or hallucination.

The reduce of total test cases could also affects the coverage as shown in column **Coverage** of table 4.7. This is due to the reduce of duplicate. When duplication occurs, Test Case Generator agent has more chance to

```

Test Case: Test Case 1
testCaseName: Navigate to Books Page via Navbar
objective: Verify that the user can navigate to the books page using the navbar and view
the initial list of books
testSteps:
  - Step 1: Open the application.
  - Step 2: Click on the 'Books' item in the navbar.
  - Step 3: Observe the list of books displayed on the page.
expectedResult: The application displays the books page with the list of the first 10
books. Each book entry shows the id, Title, Author, Genre, and Publication date.
reflect: yes
explanation: This test case directly reflects the given test scenario by verifying the
navigation to the books page through the navbar, which is the primary action
described in the scenario.

Test Case: Test Case 2
testCaseName: Pagination Navigation on Books Page
objective: Verify that the user can navigate through the book list pages using the
pagination controls
testSteps:
  - Step 1: Navigate to the books page by clicking on the 'Books' item in the navbar.
  - Step 2: Click on the 'Next' button below the list of books.
  - Step 3: Observe the list of books on the next page.
expectedResult: The application displays the next 10 books in the list. Each book entry
shows the id, Title, Author, Genre, and Publication date.
reflect: yes
explanation: This test case is a continuation of the navigation scenario, focusing on the
pagination aspect. It tests a specific functionality (pagination) that is part of
navigating within the books page, thus reflecting the given test scenario.

```

Figure 4.4: Test cases generated for scenario Navigation to a specific page of books

System	CF	CC	Coverage(%)
English Learning	55	53	96.36
Online Shopping	43	43	100
Hotel Management	36	30	83.33
Book Catalog	62	54	87.10

Table 4.7: Coverage evaluation on new test case set

generate correct test case for the same case need to be covered.

For example, in Figure 4.3, as scenario number two (Deletion Cancellation Scenario), scenario number three (User Cancels Deletion) and scenario number four (User Navigates Away Using Menu Before Confirming Deletion) give the test case generation phase three chances to generate test cases for cancelling. In the use case specification, user could either cancel by pressing cancel button or navigate to another page, to cover this use case, Test Case Generator agent must generate at least two distinct

```

Test Case: Test Case 1
testCaseName: Verify cancellation of book deletion via Cancel button
objective: Ensure that pressing the 'Cancel' button during the deletion confirmation
          dialog cancels the deletion process and retains the book record in the database
testSteps:
  - Step 1: Navigate to the book list/details page.
  - Step 2: Click the 'Delete' button next to the book intended for deletion.
  - Step 3: On the confirmation dialog, press the 'Cancel' button.
expectedResult: The book record should not be deleted from the database, and the user
                should be redirected back to the list of book records, with the previously selected
                book still present.
reflect: yes
explanation: This test case directly addresses the test scenario by verifying that the '
            Cancel' button effectively stops the deletion process, ensuring the book record
            remains in the database. It matches the scenario's requirement of cancelling a
            deletion via the Cancel button.

Test Case: Test Case 2
testCaseName: Verify navigation after cancelling book deletion
objective: To confirm that after cancelling a book deletion, navigating to other sections
          like 'Books', 'Genres', 'Authors', 'Home', or 'API' displays updated data without
          deleting the book
testSteps:
  - Step 1: Navigate to the book list/details page.
  - Step 2: Click the 'Delete' button next to the book intended for deletion.
  - Step 3: On the confirmation dialog, press the 'Cancel' button.
  - Step 4: Navigate to one of the menu items: 'Books', 'Genres', 'Authors', 'Home', or '
            API'.
expectedResult: The application should display the corresponding form with updated data,
                and the book that was subject to deletion should still be present in the database and
                visible to the user.
reflect: yes
explanation: Although this test case extends the scenario by verifying the application's
            behavior after cancelling the deletion, it still reflects the original test scenario.
            It ensures that the cancellation of the deletion process through the 'Cancel' button
            has the expected outcome not only immediately but also in subsequent interactions
            within the application.

```

Figure 4.5: Example of incorrect test case generation leads to decrease in coverage

test cases. When we leave those duplicate scenarios there, either scenario number two or three can generate two test cases for both ways of user's cancellation, which increases duplication. But as we remove scenario number three and four, Test Case Generator agent now relies on scenario number two only and has only one chance, if hallucination happens and makes only one valid and reflected test case or worse no valid or reflected test case to appear, there will be reduction in coverage. This problem is illustrated in Figure 4.5, where test case number two refer to an action of cancel by button THEN navigate, while the correct test case needs to be generated to assure the coverage is cancelling by navigation to another tab.

There are many problems in this phase, therefore we highly recommend human advisor to ensure the quality of test cases generated by this phase. In conclusion, the test case generation phase affects the final result heavily.

B. Test Case Generation with Manual Test Scenario Input:

This experiment aims to evaluate the effectiveness of Test Case Generation phase using human-written test scenarios inputs. In this evaluation, we manually explored all the use case specifications and method generated test scenarios and test cases to test and clarify the cause of errors that lead to loss of flow and condition coverage of test cases, then found ways to rewrite test scenarios to ensure minimal errors that cause loss in coverage, duplicate test cases, and test cases that do not match the use case description (incorrect test cases).

We experimented with various types of writing test scenarios, ranging from general to specific, to assess their impact on the effectiveness of test case generation. By assessing human-written test scenarios input effectiveness and compared it with our methods generated test scenarios, we can define strengths and weaknesses of our approach, as well as the factors that reduce its effectiveness, from which to propose future improvement plans.

After trying many different ways of writing test scenarios from general to detailed, we concluded that the more detailed the scenario, the more effective and stable the test case generation step will be. The number of duplicate and incorrect test cases is reduced when a detailed, clear, and specific set of test scenarios is provided for each case, as opposed to using a general set of test scenarios that covers multiple case. This is demonstrated in the table 4.8 , where the percentage of correctly generated test cases exceeds 90% when the input is a detailed test scenario that clearly represents each case to be tested, an example illustrated in Figure4.6.

In this manual test scenario example, instead of using one scenario User Cancels Deletion to demonstrate cancel case flow, we using User Cancels Deletion by Cancel Button and User Navigates Away Using Menu Before

System	Correct(%)	Duplicate(%)	Incorrect(%)
English Learning	96.6	2.27	1.14
Online Shopping	91.43	1.43	7.14
Book Catalog	90.52	3.45	6.03
Hotel Management	92.85	3.57	3.57

Table 4.8: Accuracy evaluation on test case set using human-written test scenario

1 Successful Genre Deletion
2 User Cancels Deletion by Cancel Button
3 User Navigates Away Using Menu Before Confirming Deletion

Figure 4.6: Human-Written Test Scenario Example

Confirming Deletion to make the test scenarios more specific. The test case generation results of this set of test scenarios are evaluated as 100% correct test cases and ensure coverage for all case found of the use case.

The reason why 100% correct of test case cannot be achieved is because the correct generated test case level also depends on the presentation of the use case. The logic level of the use case affects the method's ability to generate test cases. We observe that with the use case has many implicit, unclear, and general descriptions, the ability to generate incorrect test cases will also increase regardless of how specific any test scenarios are written. From there, it can be seen that the input factor also has a lot of influence on this phase.

In addition, the hallucination factor also makes it difficult for this process to generate correct test cases and increases the chances of duplicate and incorrect test cases at runtime.

With these observations, we assumed limitations of the test scenarios generated in Test Design phase is their lack of clarity and specificity. To solve this problem, we can improve the Test Design process in the future, aiming to generate more detailed test scenarios for each use case.

4.3 Comparison with Baseline Method

In this section we compared the result of our propose and the baseline method.

4.3.1 Test design phase

Usually, in those methods previously used by other researchers, the use case specification is not divided into smaller segments but is taken as a whole. Therefore, we remove the two blocks (Block 1-2) that extract conditions and flows, and keep the use case specification intact.

We then use only one prompt that force GPT to generate all scenarios directly from the use case specification. We construct the prompt for this agent with rules suitable for this agent, instructing GPT how to generate test scenarios.

This approach differs from our method, where different information is treated individually by applying a rule set tailored for each segment of the use case. Consequently, when using a fully detailed use case, our predefined rule sets cannot be applied so we have to select general rules can apply for any detail.

Finally, the baseline does not include any agent to perform the refining phase, where reviews and corrects its own responses from the previous process.

In summary, the main difference between the baseline method and our method is that the baseline uses a single agent and treats the use case as a whole, without refining or feedback mechanisms. In contrast, our method uses multiple agents and applies specific rule sets to individual segments of the use case specification, along with a refining phase to improve accuracy and consistency.

4.3.2 Test case generation phase

This phase takes the result given from test design phase and the use case specification to generate corresponding test case. We handle one scenario at a time, so the output consists only of test cases related to that specific scenario. This approach ensures a fair comparison between the baseline and our method since the problem is the same to both methods when putting full list of scenario: agents which generate test case (Block 7 Figure 3.1, Block 2 Figure 4.2) often give response where one scenario only results in one test cases, which is wrong for many cases.

The baseline method does not include a refining agent to provide feedback on whether the generated test cases accurately reflect the given scenario. This absence of feedback and correction can lead to inaccuracies in the baseline method's results.

The big difference in this phase when comparing to the our method is the absence of refining agents.

4.3.3 Comparison

Table 4.9 presents total scenarios and test cases generated by two methods for each system. Our method generates nearly three times fewer scenarios and approximately ten times fewer test cases.

System	Use case	Scenarios		Test cases	
		Ours	Baseline	Ours	Baseline
English Learning	12	74	224	101	944
Online Shopping	5	69	98	102	444
Hotel Management	12	63	204	84	709
Book Catalog	15	102	317	140	1116

Table 4.9: Total test scenarios and test cases generated from two methods

To explain for the high number of outcomes produced by the baseline method, Table 4.10 presents the ratio of three criteria within the test scenario sets for each system generated by two methods. Since the number of

test cases generated by the baseline is so many, we decided to evaluate the generated test scenarios from both methods only.

System	Correct(%)		Duplicate(%)		Incorrect(%)	
	Ours	Baseline	Ours	Baseline	Ours	Baseline
English Learning	98.65	40	0	51.67	1.35	8.33
Online Shopping	92.75	66.32	7.25	26.53	0	7.15
Book Catalog	97.06	43.53	0.98	37.85	1.96	18.62
Hotel Management	87.31	47.05	9.52	36.76	3.17	16.19
Average:	93.94	49.23	4.43	38.20	1.63	12.57

Table 4.10: Accuracy of test scenario generation

For each scenario, we classify it into one of the three stated scenario’s categories in Section ??, then sum up all scenarios in each category and calculate the percentage by dividing by the total number of scenarios generated.

Duplication problem in baseline method is caused by wording and misunderstandings, leading to test scenarios that test for each step (e.g., verify input functionality in the username text field), or test scenarios that has the same steps, the same result but just different words because treating results and actions differently (e.g., correct answer selection and green popup on correct answer), which can cause a lot of duplicate test cases.

Table 4.10 shows that the number of scenarios belong to Duplication and Invalidity category generated by baseline method is massive comparing to ours. These two categories are considered redundant, as humans need to remove illogical and duplicate scenarios. Given the substantial proportion of scenarios falling into the duplication and invalidity categories in the baseline method, we believe that the effort required to filter out usable scenarios from our results is considerably less than that required for the baseline.

As the result of the scenarios affects the generation of test case, the redundancy in test cases increases. Both methods follow the rule that each scenario generates at least one test case. Therefore, if duplicate scenarios

are not removed during the scenario generation phase, they will add to the duplication in test cases. Moreover, both methods encounter the challenge that often generates test cases that do not accurately reflect the inputted scenarios. Since the baseline method lacks an agent for validating whether the generated test cases accurately reflect the scenarios, the probability of duplicated test cases appearance increases.

We also evaluated the test case generation phase of the baseline method to compare with ours. We manually selected accurate test scenarios from the scenario set generated before to be the input for the test case generation phase of both method. By doing this, we ensure the fairness when comparing the performance of the test cases generation phase of both method.

System	Correct(%)		Duplicate(%)		Incorrect(%)	
	Ours	Baseline	Ours	Baseline	Ours	Baseline
English Learning	90.72	54.55	2.06	40.34	7.22	5.11
Online Shopping	75.31	48.72	6.17	45.29	18.52	5.98
Book Catalog	78.83	53.58	9.52	36.86	11.65	9.56
Hotel Management	68.57	53.61	17.14	38.55	14.29	7.84
Average:	78.36	52.61	8.72	40.26	12.92	7.12

Table 4.11: Accuracy of test case generation

Table 4.11 shows the accuracy of test cases generated by both method. The baseline method, even though we removed all the duplication or invalid test scenarios, still generated many redundant test cases. We took a deeper look about the reason why baseline method has high rate of creating duplication and figured out the poor performance of understanding the scenario needed to create test case for.

As mentioned before, the input for this phase is a single test scenario and the full description of use case. The main reason we do not put in the full list of the test scenario set is because the misunderstanding of task in the agent used to generate test case. When putting in the whole set, the chance of getting one single test case for one test scenario in every cases is high. The problem of getting only one test case for a test scenario is the

reduction of coverage or the combining of test cases. Figure 4.7 illustrates an incorrect test case occurred by this problem, as a test case is trying to describe two different case and connect them by using "or" in test steps.

```
testCaseName: Attempt to purchase an out-of-stock product
objective: Verify that the user cannot checkout an out-of-stock product from the shopping
          cart or product detail page
testSteps:
  - Step 1: Navigate to the shopping cart page or the product detail page of an out-of-
    stock product.
  - Step 2: Attempt to select the out-of-stock product by clicking on the checkbox before
    the item or by clicking the 'Buy Now' button.
  - Step 3: Attempt to click the 'Checkout' button.
expectedResult: The user is unable to select the out-of-stock product, and the 'Checkout'
                button is disabled or not clickable.
```

Figure 4.7: Example of a test case try to describe two separate cases

But when we use single scenario for prompt input, another problem rises for the baseline method - duplication. As shown in Table 4.11, the duplication proportion is overwhelming comparing to other categories. The biggest reason is shown in 4.8, these two test cases look normal, the problem is the agent generating these to test for a scenario named "Purchase multiple items from shopping cart". These test cases are not related to the mentioned test scenario at all.

Different from the baseline, our method has the refining agent which filter out unreflected test cases. Results from the baseline proves the important of this agent in reducing redundant test cases in test case set.

Test Case: Test Case 2
testCaseName: Purchase an item from the product detail page
objective: Verify that the user can purchase a product directly from the product detail page, including selecting options and adjusting quantity.
testSteps:
- Step 1: Perform the 'View a product detail' feature to navigate to a product's detail page.
- Step 2: If the product has multiple options, select one available option.
- Step 3: Adjust the quantity of the product if necessary by clicking the minus or plus button next to the quantity.
- Step 4: Click the 'Buy Now' button.
- Step 5: Click the 'Checkout' button.
- Step 6: Verify that the checkout process is completed.
expectedResult: The user is redirected to a page that shows a single new order of the item the user has chosen, indicating a successful purchase from the product detail page.

Test Case: Test Case 3
testCaseName: Attempt to purchase an out-of-stock item from the shopping cart
objective: Verify that the user cannot select or purchase items that are out of stock or have been removed by the seller from the shopping cart.
testSteps:
- Step 1: Navigate to the shopping cart page where the user has previously added items, including an out-of-stock item.
- Step 2: Attempt to select the out-of-stock item by clicking on the checkbox before the item.
- Step 3: Attempt to click the 'Checkout' button.
expectedResult: The system prevents the user from selecting the out-of-stock item, and the 'Checkout' button is disabled or an appropriate error message is displayed, indicating the item cannot be purchased.

Figure 4.8: Unreflected test cases for testing purchase multiple items from shopping cart

Chapter 5

CONCLUSION

5.1 Discussion of Experimental Results

The purpose of our approach is to effectively use GPT-4 for generating test cases only from use case specification. We aim to propose a method to help GPT-4 generating test case more accurately and require less human effort on advising and refining, also, the method could accept many forms of specification to take less effort in refining the input. To achieve that purpose, we ran our experiment on many formats from many fields to test the ability of GPT-4 in capturing important information and generating manual test cases.

Given by result from Chapter 4, our method could enhance the process of using GPT-4 to automatically generating test cases for manual testing from use case specifications. By adding refining agents to evaluate the response, our propose improve the quality of test cases generated and reduce human effort.

We evaluate our method based on two primarily metrics, the accuracy of test cases and the test coverage. We want tester could use the test case immediately, with less effort to remove redundant test cases or supplement important missing test case. In conclusion, the number of invalid test cases are small, tester could possibly need little effort to remove these test cases while testing the system. But the outcome also proves that much larger

effort is needed to remove duplicate test cases.

We also want our propose could apply for many formats, from simple to complex use case specification. Although, the result proves that the more simply the use case is described, the more worse the final result would be. This suggests that while our method is effective, it performs better with more detailed and structured use case specifications.

Overall, our proposed could possibly using GPT-4 for generating test cases from use case specifications though there is still many limitations need to overcome.

5.2 Limitations

The proposed approach has several limitations. One limitation of our current approach is the inability to generate test data for the produced test cases. Time is one reason, other reason is we still not yet come up with fine approach for detecting test cases need test data, and test data could test complex cases, edge cases and boundary cases.

Our agents all use Zero-Shot Prompting to proceed. The reason why we could not use Few-Shot Prompting and Chain-of-Thought Prompting efficiently enough to put it in usage for our propose is because we have not yet to find suitable presentation for examples. Since our proposal aims to be workable with many formats, choosing a specific format as our example would result in very poor outcomes for other formats. Moreover, deciding which use case to use for presentation is also problematic. If a simple use case is chosen, complex use cases could result in poor test cases. Conversely, using a complex use case as an example might not be applicable to simpler cases, leading to inconsistency and reduced reliability across varied use case specifications.

Other limitation is the exist of redundant test cases. Even though we have refining agents act as removers for redundant result, these agents still cannot perform well, especially with specifications with unclear or simple

description. Duplication is still a huge problem in our method.

Furthermore, although we state that our method could accept many formats with important events are written in flows, the difference in format still affects the final result heavily. The lack of datasets from real-world applications or use case specifications from businesses, provided by professional Business Analysts also makes our propose hard to be comprehensively evaluate.

Our study also lacks a comparison with other methods that also use LLMs with different approaches or different models. This limitation affects the overall study by not providing a benchmark to gauge the relative effectiveness and efficiency of our method. Without this comparison, it is challenging to determine if our method has improve existing methods or not in the field of using LLMs to generate manual test cases.

5.3 Future Work

To address these limitations, future work should focus on several key areas.

First, trying more prompting techniques and finding suitable examples could help improve the adaptability and performance of our approach. Exploring more prompting techniques could help define which technique is most suitable for each task in our method. Also, finding a strategy to select appropriate examples that generalize well without compromising on specificity for different scenarios is crucial.

Second, enhancing the refining agents to better identify and remove redundant test cases. By improving these agents, we could gain result with less redundant test cases, this lead to reduction in human efforts needed to review and refine test cases.

Thirdly, addressing the challenge of generating test data for test cases is critical for comprehensive testing coverage. Developing mechanisms to automatically generate test data, covering complex, edge, and boundary

cases, would enhance the effectiveness of the generated test cases.

Fourthly, we should research more to see the current trends in using LLMs for manual test cases generation. We could compare our propose with other related methods and find our advantages, disadvantages for improvement.

Lastly, to validate and enhance our approach in real-world settings, we should experiment our propose on specifications written by industries or professional business analysts. This would provide valuable insights into the practical applicability and performance of our method in diverse operational environments.

Bibliography

- [1] Q. Zhang, C. Fang, Y. Xie, *et al.*, “A survey on large language models for software engineering,” *arXiv preprint arXiv:2312.15223*, 2023.
- [2] A. Fan, B. Gokkaya, M. Harman, *et al.*, “Large language models for software engineering: Survey and open problems,” *arXiv preprint arXiv:2310.03533*, 2023.
- [3] 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.
- [4] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, “Automatic generation of system test cases from use case specifications,” in *Proceedings of the 2015 international symposium on software testing and analysis*, 2015, pp. 385–396.
- [5] R. Gröpler, V. Sudhi, E. J. C. García, and A. Bergmann, “Nlp-based requirements formalization for automatic test case generation.,” in *CS&P*, vol. 21, 2021, pp. 18–30.
- [6] J. Fischbach, J. Frattini, A. Vogelsang, *et al.*, “Automatic creation of acceptance tests by extracting conditionals from requirements: Nlp approach and case study,” *Journal of Systems and Software*, vol. 197, p. 111 549, 2023.
- [7] T. Rahman and Y. Zhu, “Automated user story generation with test case specification using large language model,” *arXiv preprint arXiv:2404.01558*, 2024.

- [8] M. D. Haiderzai and M. I. Khattab, “How software testing impact the quality of software systems?” *IJECS*, vol. 1, no. 2, pp. 05–09, 2019.
- [9] P. Kroll and P. Kruchten, *The rational unified process made easy: a practitioner’s guide to the RUP*. Addison-Wesley Professional, 2003.
- [10] A. Cockburn, “Structuring use cases with goals,” *Journal of object-oriented programming*, vol. 10, no. 5, pp. 56–62, 1997.
- [11] H. Touvron, T. Lavril, G. Izacard, *et al.*, “Llama: Open and efficient foundation language models,” *arXiv preprint arXiv:2302.13971*, 2023.
- [12] D. Driess, F. Xia, M. S. Sajjadi, *et al.*, “Palm-e: An embodied multimodal language model,” *arXiv preprint arXiv:2303.03378*, 2023.
- [13] OpenAI, J. Achiam, S. Adler, *et al.*, *Gpt-4 technical report*, 2024. arXiv: 2303.08774 [cs.CL]. [Online]. Available: <https://arxiv.org/abs/2303.08774>.
- [14] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, “Improving language understanding by generative pre-training,” 2018.
- [15] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [16] T. Brown, B. Mann, N. Ryder, *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [17] M. Abu-Haifa, B. Etawi, H. Alkhatatbeh, and A. Ababneh, “Comparative analysis of chatgpt, gpt-4, and microsoft copilot chatbots for gre test,” *International Journal of Learning, Teaching and Educational Research*, vol. 23, no. 6, pp. 327–347, 2024.

- [18] K. I. Roumeliotis, N. D. Tselikas, and D. K. Nasiopoulos, “Llms in e-commerce: A comparative analysis of gpt and llama models in product review evaluation,” *Natural Language Processing Journal*, vol. 6, p. 100 056, 2024.
- [19] H. Rehana, N. B. Çam, M. Basmaci, *et al.*, “Evaluation of gpt and bert-based models on identifying proteinprotein interactions in biomedical text,” *ArXiv*, 2023.
- [20] A. Koubaa, “Gpt-4 vs. gpt-3.5: A concise showdown,” 2023.
- [21] J. Heumann, “Generating test cases from use cases,” *The rational edge*, vol. 6, no. 01, 2001.
- [22] S. Feng and C. Chen, “Prompting is all you need: Automated android bug replay with large language models,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.
- [23] A. Madaan, N. Tandon, P. Gupta, *et al.*, “Self-refine: Iterative refinement with self-feedback,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [24] A. F. Akyürek, E. Akyürek, A. Madaan, *et al.*, “Rl4f: Generating natural language feedback with reinforcement learning for repairing model outputs,” *arXiv preprint arXiv:2305.08844*, 2023.
- [25] W. Yu, Z. Zhang, Z. Liang, M. Jiang, and A. Sabharwal, “Improving language models via plug-and-play retrieval feedback,” *arXiv preprint arXiv:2305.14002*, 2023.
- [26] Z. Ji, T. Yu, Y. Xu, N. Lee, E. Ishii, and P. Fung, “Towards mitigating llm hallucination via self reflection,” in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 1827–1843.