Universiteit Leiden

Master Computer Science

The Impact of AI on Enhancing Software Testing

Name:	Xiaolin Gu
Student ID:	s3790398
Date:	20/05/2025
Specialisation:	Advanced Computing Systems
1st supervisor:	Marcello M. Bonsangue
2nd supervisor:	Joost Visser

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

1 Introduction 4

2 AI in Documenting Software Testing 5
2.1 AI integration in Documentation Overview . . . 6
2.2 Methodology . . . 6
2.2.1 Data Privacy . . . 6
2.2.2 AI Input Processing . . . 6
2.2.3 Output Validation . . . 8
2.3 Experiments . . . 8
2.4 Analysis of the Results . . . 10

3 AI in Test Case Design 12
3.1 AI integration in Test Case Design Overview . . . 12
3.2 Methodology . . . 12
3.2.1 APIs Utilized in Experiments . . . 13
3.2.2 Upload Wiki . . . 13
3.2.3 Send Request Description Generated by SwisscomGPT . . . 14
3.2.4 Study With Different Request Accuracy . . . 15
3.3 Experiments . . . 15
3.3.1 Upload Wiki . . . 15
3.3.2 Prompt Generated by SwisscomGPT . . . 16
3.3.3 Test Cases Designed by SwisscomGPT Chat . . . 17
3.3.4 Test Cases Designed by Cursor Chat . . . 17
3.3.5 Request Accuracy . . . 17
3.4 Analysis of the Results . . . 19
3.4.1 Test Case Validation . . . 19
3.4.2 Invalid Test Cases Overview . . . 19
3.4.3 Test Cases Expectation . . . 19
3.4.4 Test Cases Designed By Different Methods . . . 20
3.4.5 Test Cases Designed Using Different Request Accuracy . . . 22

4 AI for Automated Test Code Generation 23
4.1 AI integration in Test Code Generation Overview . . . 23
4.2 Methodology . . . 23
4.2.1 Exploration of Cursor Capabilities . . . 23
4.2.2 Project Structure . . . 24
4.2.3 Design of Test Code Generation Prompts . . . 25
4.2.4 Validation of Generated Test Code . . . 27
4.3 Experiments . . . 28
4.3.1 Manual generation . . . 28
4.3.2 Code generated by GPT-4o . . . 28
4.3.3 Code generated by Claude-3.5-Sonnet . . . 31
4.4 Analysis of the Results . . . 34

5 Broader Discussion 37
5.1 Interplay Between Human and AI Contributions 37
5.2 Privacy and Security Concerns in AI-assisted Testing 37
5.3 Handling Multilingual and Outdated Data 37
5.4 Specialized AI Models for Niche Applications 38
5.5 Integration of AI Tools into Industry Settings 38
5.6 Potential Risks of AI-generated Testing Beyond Technical Aspects 38
5.7 Generalization Beyond the EsBill Project 39

6 Conclusion 39

A APIs Utilized in Experiments 42

B Example AI response for test case design 43

C Test cases designed by AI 50

Abstract

Software testing is a crucial phase in software engineering, essential for ensuring the quality and reliability of complex systems. With the rise of increasingly sophisticated AI tools, AI-assisted software testing has gained momentum, offering promising support in automating and enhancing testing processes. This study investigates the synergy between AI and human testers, focusing on how AI can handle testing-related tasks and how human expertise complements its limitations. Using the EsBill project at Swisscom as a case study—a real-world enterprise billing system—the research evaluates the effectiveness of tools like SwisscomGPT and Cursor in retrieving legacy documentation, designing test cases, and generating test code. The experiments were assessed by experienced software testers from the EsBill team. Key research questions address the extent to which AI can process legacy knowledge, the impact of model and prompt variations on test case quality, the comparative performance of AI-generated versus human-written test code, and the role of human oversight in the AI-assisted workflow. While the findings are contextual to the EsBill project, they offer valuable insight for other enterprise environments aiming to integrate AI into their testing practices.

1 Introduction

Testing is a very important stage in software engineering as it guarantees the quality of the project. As software systems become increasingly complex, the importance of automating software testing continues to grow. With the rapid development of AI tools in recent years, more and more software projects have introduced AI as a support, especially in software testing stage. AI-assisted software engineering is not a novel idea. Many studies have explored how possible this idea can be. The study by Ahmad et al. [1] suggested that ChatGPT can complete the work as an architect, but human oversight and decision support for collaborative architecting is still required. Bareiß et al. [2] studied how Codex performs in code mutation, test oracle generation from natural language documentation, and test case generation tasks. This study focuses on AI-Human Synergy, exploring AI's ability to deal with testing knowledge and tasks, and how humans supplement AI's work. We conducted the experiments on the basis of the EsBill project from Swisscom. Swisscom is one of the leading telecommunications and IT companies in Switzerland. EsBill is the billing system of B2B and creates invoices for Swisscom's Business Customers on the one hand and serves as an application and information system for employees and the peripheral systems on the other. EsBill mainly uses Swisscom Wiki to store knowledge. Some other document is stored in sharepoint and Microsoft Word. The project is developed using Java and Python, and code is pushed to GitLab and uses postgresDB as database tool. We used SwisscomGPT and Cursor to conduct experiments for testing documentation, test case design, and test code generation. The results of the experiment were evaluated by experienced EsBill team testing engineers. It is important to note that the methods and findings in this study are based on the specific context of the

EsBill project and are not intended as general-purpose solutions. However, its methods and findings are also inspiring for other enterprise projects.
The main research questions of this study are as follows.

- RQ1: To what extent can LLMs retrieve information from legacy documentation to support software testing tasks?
- RQ2: How do different AI models and input detail levels influence test case generation?
- RQ3: How does AI-generated test code improve performance and usability over human-written code?
- RQ4: How can humans complement and collaborate in the AI-assisted testing process?

The thesis contains 7 chapters. Chapter 2 shows the relation of this study to existing LIACS courses. Chapter 3 introduces the application of AI in documenting software testing, focusing on analyzing SwisscomGPT's processing capabilities and limitations for structured and unstructured documents. Chapter 4 explores and compares the performance of different AI models in designing API test cases. Chapter 5 studies the quality and efficiency of AI-generated test code compared to manually written code. In chapter 6 we discuss the synergy between AI and humans in the testing process, pointing out the shortcomings of current technology and directions for improvement. The final chapter summarizes the experimental findings and proposes suggestions for industry applications and future research.

2 AI in Documenting Software Testing

There are multiple challenges in documentation from software projects. Normally, different documents will be stored on different platforms. For example, codes are usually stored on GitLab or GitHub. Project management documents are mainly stored on Jira. Design and API docs will be stored on Sharepoint or wiki. Many of these documents may be incomplete or outdated, particularly those from older software projects. Different types of documents contain distinct terminologies and data formats that require preprocessing.
Documents for EsBill include API docs, requirement docs, architecture docs, design docs, development docs, testing docs, project management docs, etc. These documentations are used by people from EsBill team, external teams and upstream and downstream teams. A key challenge in data collection for the EsBill project is its age, as it has been in operation for over 20 years. We need to validate outdated data in documents to prevent incorrect AI-generated responses in the future. Additionally, there is a huge amount of knowledge on many different platforms such as Wiki, Sharepoint. Wiki alone contains more than a hundred pages. We note that all this knowledge is written in English or German, making it harder for project participants to find the information they need.
Swisscom developed our own AI called SwisscomGPT, which is an instance from OpenAI. In SwisscomGPT, we don't need to worry about privacy issues, as data uploaded will not be fed to public AI. Meanwhile, it is still a new product that is not widely used in Swisscom, so it would also be valuable for the company to explore the potential of this tool. Considering these, we decide to choose SwisscomGPT as our tool.
In this chapter, we will use SwisscomGPT to study how AI can help engineers organize documents and collect the information needed from a huge number of documents, and how AI performs in this task.

2.1 AI integration in Documentation Overview

Previous works have explored different technologies to use AI to assist information retrieval. SimplyRetrieve[3] is an open-source RCG platform that leverages a Private Knowledge Base Constructor and a Retrieval Tuning Module. A recent study[4] investigated a new approach to enrich document representation during indexing using generative AI and demonstrated that generative AI can efficiently fill the knowledge gap in document representation. KE4IR system[5], which utilizes a document retrieval approach that takes advantage of Linked Open Data and Knowledge Extraction techniques, has better retrieval performance than exploiting only textual information. Knollmeyer et al.[6] enhanced RAG systems by integrating a Knowledge Graph (KG) constructed from the document structures to break limitations when users' questions extend beyond the text content of the documents.

2.2 Methodology

2.2.1 Data Privacy

EsBill is a project deals with B2B customers, so data privacy is an important issue.
Since all externally accessible data outside Switzerland has been anonymized, we do not need to address additional data privacy concerns.
To test how AI works on different types of documents, we fed AI with Wiki pages containing tables and images, and the database structure of esBill databases. Since SwisscomGPT only accepts documents in PDF form, we export wiki pages in PDF.
For database information, we exported the command of creating the command by selecting tables directory and clicking ”SQL Scripts”, ”Generate DDL to Clipboard”. We deleted all grant commands and transferred the DDL to PDF. By processing this PDF, the AI can extract information about the structure and fields of the database tables.

2.2.2 AI Input Processing

We created an AI instance named “EsBill Helper” and uploaded all knowledges from esBill wiki to this instance.
To improve the AI's performance in satisfying our needs, we conducted a preliminary study and held internal meetings to gather team members' opinions on the rules the AI should follow during information retrieval. Based on these discussions, we formulated the following five prompts and applied them to our AI instance:

1. Use Documentation as the Primary Source:
When responding to queries, base your answers strictly on the information provided in my documentation. Avoid using external or general knowledge unless specifically instructed to do so.

This prompt ensures that the AI's responses are strictly grounded in the documentation from the team. The goal is to avoid introducing irrelevant or potentially misleading external information that could conflict with the project requirements.

2. No Fabrication of Information:
Do not generate information that isn't explicitly found in my documentation. If you cannot find the necessary details in the provided material, avoid making assumptions or fabricating

data.

This prevents the AI from making up information that is not present in the documentation, as fabricated information can lead to work errors. EsBill team needs AI to act as an accurate reflection of the documentation, not as a source of unverified information. If the documentation lacks the necessary information, the AI should not guess or fabricate answers.

3. Proposal of Alternatives:
If a question is asked that is not covered by the documentation, notify me by stating, "This information is not available in the provided documentation." Afterward, you may suggest a potential answer or approach, but only if you make it clear that the suggestion is not based on the documented material.

This prompt ensures transparency about the AI's limitations while still providing users with potentially helpful ideas. If a query falls outside the scope of the documentation, the AI should notify the user that the information isn't available. It may then suggest alternatives, but must clarify that they are not derived from the documentation. It balances accuracy with creativity.

4. Transparency:
Always distinguish between responses based on the documentation and those that are speculative or outside the scope of the provided information. Use phrases like "Based on your documentation" or "This suggestion is outside the documented material" when appropriate.

We need to know which parts of the AI's output are verifiable within the provided knowledge and which are speculation. The AI should clearly differentiate between answers based on the documentation and speculative or external suggestions. Using explicit phrases such as "Based on your documentation" ensures clarity.

5. Handling Uncertainty:
If you are uncertain about the answer based on the documentation, explicitly say so before proposing any further steps or alternative information.
If you get the request to display some API request or some details of API request, please display the HTTP request details in the following structure:
URL: {request_url}
Method: {http_method}
Headers: {request_headers}
Body: {request_body}"
When providing this instruction, replace {request_url}, {http_method}, {request_headers}, and {request_body} with the actual data you want GPT to format for you.
If you need GPT to generate or construct an HTTP request based on certain criteria you provide, you can give it detailed information about what you're looking for. Here's an example instruction for that scenario:
"Generate an HTTP POST request that includes the necessary headers for a JSON content type and an authentication token. Fill in the body with data for a new user registration. Please display the request as follows:
URL: {provide specific endpoint here}
Method: POST
Headers:
Content-Type: application/json

> Authorization: Bearer {your_auth_token_here}
> Body:
> Replace placeholders such as {provide specific endpoint here} and {your_auth_token_here} with real values. The AI will then structure the HTTP request in a readable format you've provided.
> jsonCopy code{”username”: ”new_user”,”email”: ”newuser@example.com”,”password”: ”SecurePass123!”}“”

AI should acknowledge when it is unsure or when the documentation lacks the necessary information, explicitly stating this before proposing any solutions. This helps avoid overconfidence in the AI's responses, ensuring that users aren't misled into assuming incorrect information is definitive.

For the database knowledge, we created another AI instance named ”DB Helper”. Prompts are the same as ”EsBill Helper” without the API form restrictions in the last instruction (5. Handling Uncertainty).

During the data feeding process, we recorded the number and size of documents and measured the time required to assess the effectiveness of AI-assisted documentation management. We also tracked the upload success rate and analyzed any files that failed to upload.

2.2.3 Output Validation

To validate the capability of AI to understand knowledge and collect information, we will ask the skilled employees of the esBill team who have worked with this knowledge for years to design some questions based on the uploaded documents. By the ground truth created by them, we were able to make the analysis more theoretically rigorous. Questions will be generated based on different documents, from different data type such as table in the documents. We will analyze the correct rate of AI's answer. For the wrong answers, we will further analyze the reasons and think about solutions to train AI to better answer the question.

2.3 Experiments

We fed AI with EsBill system documents. These documents come from three different places in EsBill wiki: pages under How-to-guides page which includes trouble shooting for EsBill testing work; pages under Schnittstellen page which includes introduction of interfaces about account and bill; pages under Knowledge Sharing page which includes knowledge of different components in EsBill system. To upload all this wiki pages to SwisscomGPT, we exported these pages to PDF. There were in total 78 documents, with a total size of 67.8MB. 25 documents among them are written in Germany and remains are written in English. It took around 1 hour to export all these PDFs.

The documents needed to be uploaded one by one. We had to upload the documents individually, which took approximately ten minutes. Once uploaded, the AI processed them within seconds. During the upload process, one page (”Process eInvoices — Transfer files to ECC”) disappeared after we clicked the upload button. This page included a screenshot displaying the ECC ”Rechnungsnummer” and eight steps for transferring files to ECC. We decided that pictures inside are no important and copied all texts in one PDF to upload this page.

For database knowledge, there are in total 20 PDFs. All PDFs were uploaded successfully in approximately ten minutes.

We created some questions related to the pages uploaded to test how this AI instance worked. The experiment results are as follow:

Source	Question	Result	Solution
Wiki	How to remove UC4 jobs	Answered correctly	
	What is the main flow to process file with UNISS job (create OTC)	Answered correctly	
	Could you please write all informations about esBill account?	Answered knowledge only listed in Acount page	We changed the question to "Tells me everything about esbill accounts without listing the section of account page" and AI answered the question comprehensively
	Which user group in esBill do you know?	Answered correctly	
	Could you provide all code and meanings for MESSAGE_STATE_CD in TOMESLG?	AI can't provide knowledge about this	We observed that the AI struggled to process information embedded within tables. So we copied the contents inside the table in a new PDF and uploaded it again. After reuploading AI answered the question correctly
	In which directory has to be dateMutatorOrderFile for layouts?	Answered correctly	
	Do we need to adjust the configuration file for datemutator?	Answered correctly	
	How to do rater step by step?	AI didn't reply	We changed the question to "How to work with Rater? step by step" "how to do Rater? step by step" "how to do Rater step by step" "how to do rater? step by step" and AI answered all these questions correctly
	List all the API requests you know related to esBill	Replied "no specific API requests related to esBill have been mentioned"	
	Is MsgId mandatory field in the input of the searchEMailAddress?	Answered correctly	

Source	Question	Result	Solution
Database	Can you provide all tables that have contr_id?	This information is not available in the provided documentation. The provided document does not contain any information about tables with a "contr_id" attribute.	We asked the same question again. See AI's response in the next row
	Can you provide all tables that have contr_id?	I do not know. The provided document does not mention any tables with a "contr_id" attribute.	We asked the same question again. See AI's response in the next row
	Can you provide all tables that have contr_id?	AI answered 10 correct tables, but that's not all tables that includes contr_id.	We continued to ask AI about this issue. See AI's response in the next row
	Are there any more tables that contain contr_id?	AI repeated 7 tables from the previous answer and added two more tables, but these are still not all tables	We suggested that this is because the database is too complex, so we selected one file containing field contr_id and deleted all the database privilege management commands inside as data source to test again
	Can you provide all tables that have contr_id?	AI gave 9 tables which contain contr_id, however there are 15 tables contain contr_id in total	We continued to ask AI about this issue. See AI's response in the next row
	Are there any more tables that contain contr_id?	AI provided the same 9 tables as previous answer and added "I can't find any further instances of 'contr_id' in other tables within this document". We gave up this experiment	

2.4 Analysis of the Results

To better evaluate the result, we introduce precision, recall, F1-score, average attempts to correct answer and one-shot accuracy for analysis. We define concepts as follow:
True Positive(TP): The number of questions AI answered correctly finally(for wiki-based knowledge) / The number of tables that the AI answered correctly finally(for database knowledge)
False Positive(FP): The number of questions AI tried to answer but gave wrong or irrelevant answer(for wiki-based knowledge) / The number of tables that the AI answered incorrectly(for

database knowledge)
False Negative(FN): Questions that AI did not answer or refuse to answer(for wiki-based knowledge) / The number of tables that the AI missed(for database knowledge)
Average Attempts to Correct Answer: The average number of times AI required to answer the question correctly
One-shot Accuracy: The proportion of questions that AI answered correctly on the first try
And we calculate precision, recall and F1-score using formulas:

$$\textbf{Precision} = \frac{TP}{TP + FP}$$

$$\textbf{Recall} = \frac{TP}{TP + FN}$$

$$\textbf{F1} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

During the validation phase for wiki-based knowledge, the AI correctly answered 6 out of 10 questions. 2 unanswerable questions were answered correctly after a further process of documents. 1 question was not answered well at the beginning but got a better answer after revising the question. For database knowledge, AI answered 9 tables and missed out 6 tables. It failed to answer all the tables at the end of the experiment.

	Precision	Recall	F1-Score	Average Attempts to Correct Answer	One-shot Accuracy
Wiki-based Knowledge	90%	100%	94.74%	1.4 times	60%
Database Knowledge	100%	60%	75%	0 times	0%

Table 1: Performance of AI in Documenting Software Testing

For wiki-based knowledge, AI can extract correct information in most cases. It showed good modifiability and answered correctly after we modified the question. But we found that SwisscomGPT cannot process knowledge within tables and images. This might not be general for all AIs, since the original ChatGPT can analyze tables and pictures. For SwisscomGPT, we developed methods to convert this information into a format that could process by extracting the information in the tables and pictures and converting it to texts. This highlights the importance of verifying the types of information an AI system can process before relying on it for retrieval, ensuring that critical data is not overlooked.
However, AI performed relatively poorly when dealing with database knowledge. It had a low precision and could not answer all tables even after we simplified the document. Database often contain more structural and formatted data. These findings suggest that we need to further refine database knowledge by providing contextual information, such as table descriptions, attribute meanings, and field explanations, to improve AI comprehension. In addition, AI in this stage might not be eligible for handling database knowledge in a real software project, as database in actual project are always large and complex.

3 AI in Test Case Design

In esBill we work mainly on API testing. Our experiments focused on testing APIs from esBill and analyzing how AI works during this process. We tested 6 APIs including 3 GET requests and 3 POST requests. We used SwisscomGPT and Cursor Chat to generate test cases based on the request description and compared their performances. For SwisscomGPT, we conducted the experiments by uploading the API wiki pages directly and by directly uploading the API wiki page and sending it a description of APIs in text.

3.1 AI integration in Test Case Design Overview

Today, the use of LLMs in test case design is a widely researched topic. Many studies have explored methods and developed test case design tools that integrate LLM capabilities. EvoSuite[7] is one of the most well-known tools for test case generation. Written in Java, it can automatically generate test cases with assertions for various classes. Gao et al. focused on prompt engineering and introduced MAPS, an LLM-tAilored Prompt generation method, to improve test case generation using LLM[8]. ChatUniTest[9] harnesses the capabilities of LLM and uses a generation-validation-repair mechanism to improve the quality of generated test cases. JCrasher[10] generates test code based on Java class type information, while SUSHI leverages symbolic execution to generate test cases.
All of these studies explored different methods for AI to help with software testing and have laid a solid foundation for this area. However, our research will not focus on exploring and evaluating these tools. Instead, we would leverage SwisscomGPT and Cursor to assist in the test case design task. We will explain these tools and their value for research in the next section.

3.2 Methodology

We kept using SwisscomGPT in this study, as it is an AI developed and used only by Swisscom and is not widely used by developer now, making it valuable for research.
There is another AI tool that is now widely used in IT industry called Cursor. Cursor is a code editor integrates AI-assisted functions such as code completion, real-time suggestions, and automatic test case generation. It is built on top of Visual Studio Code (VS Code), thus it has interfaces and functionalities similar as VS Code. One of Cursor's key features is test code generation, which we would study about in the next chapter. For test case design, the Chat function in Cursor is a good assistance for this task. It provides an interactive chat interface that the developer can chat with AI to receive coding assistance from AI.
In conclusion, Cursor Chat is more professional in coding and software development, while SwisscomGPT is a more general AI assistance. To ensure a more streamlined and integrated workflow, we decided to choose Cursor as another AI tool and compared their performance during the study.
We generated test cases using the following three approaches:
1. Uploaded the wiki page of tested API to SwisscomGPT and asked it to generate test cases.
2. Send complete descriptions to SwisscomGPT with all request details of the API tested from Wiki in chatbox to generate test cases.
3. Sent complete descriptions to Cursor Chat with all request details of the API tested from wiki in chat to generate test cases.

The quality of the generated test cases was evaluated by experienced testers at esBill. Using these three different approaches, we compared the effectiveness of API description uploads and evaluated which AI model performed better.
Additionally, we conducted a small experiment to analyze AI performance under different request accuracy levels and included the results in the analysis section.

3.2.1 APIs Utilized in Experiments

To conduct the experiments in this chapter, we selected four APIs that we had previously tested during daily work. Two of the APIs, Read Contract History and Read Ust-Id, use the GET method, while the remaining two, Change Address V2 and /ustId/v1, use the POST method. Detailed descriptions for these APIs are listed in Appendix A.

3.2.2 Upload Wiki

Similar to the operations described in the Documentation section, we created a SwisscomGPT instance called Test Case Designer (Upload) and uploaded the wiki pages for all requests. We gave this instance instructions as follow:

Please provide me test cases by completing the following task and ensure the response follows the specified format:
Understand the APIs in the file and provide test cases.
Test Case ID
- **Test Case Description**: [Describe what function of the API is going to be tested here]
- **Test Steps**: [Include test data here, don't provide test data that parameter type is HEADER]
- **Expected Result**: [What result do you expect?]

—

Important Notes
- Ensure all sections (Test URL and Test Cases) are clearly formatted.
- The generated content must align closely with the language and structure above.

Build more test cases like this one to cover varied scenarios and achieve complete coverage.

This part ensures that each test case follows a consistent format, so the context and purpose of each test case can be easily understood and test steps can be clearly followed to execute. By specifying which parameters to include or exclude in test data (e.g., avoiding HEADER parameters in the test steps), the instruction ensures clarity in the test case setup and focuses on the relevant parameters, making test data management simpler and more efficient. At the end of the instructions, we aimed to improve API test coverage by generating test cases for various scenarios.
When asked SwisscomGPT to design test cases for me, we provided the name of API, accountId/contractId in the prompt. For example:

Give me test cases for Read Ust-Id API with accountId 13004204

3.2.3 Send Request Description Generated by SwisscomGPT

To send request description to SwisscomGPT and Cursor Chat, we first created a SwisscomGPT AI instance called Test Case Prompt Generator and uploaded wiki pages of all requests. The final version of instructions after multiple experiments are as follow:

> Write detailed instructions for designing comprehensive test cases for the specified API.
> Follow these requirements:
> API Details:
> HTTP Method: Only use POST or GET.
> Endpoint: Provide the exact endpoint, including the prefix https://stesbonline01:<port>/JaxRsWeb.
> Parameters:
> List all parameters, including headers, path, query, JSON payload, and nested fields. For each parameter, provide:
> Name: Include nested JSON fields explicitly.
> Type: Data type (e.g., String, Integer, ListEnum>).
> Mandatory/Optional: Clearly specify whether required or optional.
> Max Length: For string fields, specify maximum lengths.
> Description: Briefly explain the purpose of the parameter.
> Examples: Include realistic sample values for each parameter.
> Expected Output:
> Success Response:
> HTTP Code 200 with an example JSON response.
> Error Responses:
> List possible error scenarios, including invalid/missing parameters or unsupported values, with sample codes and messages.

We gave the instance some requirements based on previous experiments. For HTTP methos, since AI often provided PUT instead of POST, we restricted the HTTP method it could provide. For endpoint, we gave it the prefix in order to receive a complete API endpoint. In the parameters part, we asked it to provide all kinds of parameters mentioned in the wiki to prevent it from omitting any parameters and informations. Additionally, we set rules for output part for generating detailed output information for coding part.

> Test Case Design Requirements: Coverage Goals:
> Test all valid scenarios with complete and minimal data.
> Test all combinations of optional parameters.
> Include invalid and edge cases (e.g., max lengths, unsupported values).
> Cover security aspects (e.g., unauthorized access).
> Include performance tests (e.g., high-volume requests).

The purpose of these instructions is to ensure that using the generated test case prompts, testing code can fully cover the API's functionality, boundaries, error handling, security, and performance, which help improve the quality of test cases.

> Structure:
> Test cases must include:
> Test Case ID: Unique identifier.

Description: Purpose of the test.
Preconditions: Setup or requirements.
Steps: Detailed execution steps.
Input Data: All parameter values.
Expected Results: Define the output for success or failure.

Same as the instructions in last sections, these make the test case prompts clear and easily and easier to understand.

Critical Instructions to Avoid Errors: Only use POST or GET as the HTTP method.
Include all JSON payload parameters, including nested fields.
Provide the full endpoint without omissions.
Ensure parameter details are complete and accurate.
Follow the format for inputs, outputs, and test case structure.

At the end of the instructions, we emphasized my requests regarding the AI's mistakes and omissions during the experiment again to ensure the AI fully understand my requirements.

3.2.4 Study With Different Request Accuracy

To change the request accuracy, we assumed the request description generated by SwisscomGPT Change address v2 API in the previous section is of 90% accuracy, and asked AI to generate other two request descriptions with 30%, 60% accuracy.
A request with 90% accuracy contained all necessary details, including parameter lists, exact field formats, and suggested input examples. It provided guidance on security and performance testing, and took account boundary conditions and error handling into consideration. A request with 60% accuracy also contained all parameters and provided some suggested input examples, but field description is brief. It mentioned security and performance testing but did not provide specific methods. It also did not explicitly stress all boundary conditions. A request with 30% accuracy contains only basic test steps and a limited number of test points. It didn't list all parameters and their meanings in detail. The descriptions of error handling, security, and performance testing were vague, and they lacked verification for boundary conditions.
Same as other test case design experiments, we sent these prompts to SwisscomGPT chat to obtain test cases.

3.3 Experiments

To enhance readability, we provided one API test case as example and a table a table listing the test cases descriptions for all APIs in each sections.

3.3.1 Upload Wiki

AI responded all APIs first with a sentence containing the name of API and the accountId/contractId we specified, then followed by 4 different test cases. Each test case contains a description, test data, test steps, and expected results. Example AI response can be found in Appendix B.

3.3.2 Prompt Generated by SwisscomGPT

In 4.2.3, we created a SwisscomGPT instance called Test Case Prompt Generator to generate prompts that asks AI to design test cases. To generate these prompts, we specified the name of the API and accountId/contractId, and asked the instance to give me the prompt. For example:

> Write a prompt to design test cases for the API "Change Address v2" with accountId=13004204.

Test Case Prompt Generator would give me prompts beginning with the aim of testing.
After obtaining the prompt from this instance, we sent the prompt to SwisscomGPT and Cursor Chat and asked them to give me detailed test cases based on information inside the prompts. Follow are test steps, including HTTP method, original URL, and headers. Then it listed input data, including HTTP method, URL with specific accountId/contractId, headers, and JSON payload if needed. Next in expected results part, the prompts described what would happen when a request executed successfully, and under which circumstances the request would fail. In the end, the prompts stressed some additional testing scenarios such as negative testing, security testing, performance testing, etc.
Example prompt is as follow:

> Create a test case to evaluate the "Change Address v2" API functionality which is intended to modify the address associated with a specified accountId.
> Steps:
> 1. Establish a secure connection with the server.
> 2. Prepare a POST request for the given endpoint: `https://stesbonline01:<port>/JaxRsWeb/account/{accountId}/address/v2`, where accountId must be replaced with "13004204".
> 3. Include necessary headers in your request:
>   * endUserId (Mandatory): It can be any String. For instance, "tgdgalu1".
>   * correlationId (Optional): It can be any String like "8c38500e-2de8-11ed-a261-0242ac120002".
> 4. Construct a JSON Payload for your POST request including:
>   * Mandatory fields: name1 and cityTown.
>   * Optional fields (if needed): street, houseNo, zip, and countryCd.
> 5. Send the POST request to the server.
>
> Input Data:
>   * HTTP Method: POST
>   * URL: `https://stesbonline01:<port>/JaxRsWeb/account/13004204/address/v2`
>   * Headers:
>   * correlationId - "8c38500e-2de8-11ed-a261-0242ac120002"
>   * endUserId - "tgdgalu1"
>   * JSON Payload:
>   * name1 - "John Doe"
>   * cityTown - "New York"
>   * street - "235 E 46th St"
>   * houseNo - "235"

```
        * zip - "10017"
        * countryCd - "US"
Expected Results:
    1. If the request is successful, you'll receive an HTTP Status Code 200 along with a
JSON
        response which should contain the new addressId.
    2. If the accountId is not valid, or does not exist, or if mandatory parameters are
missing, you'll
        receive an appropriate error response.
Additional Testing Scenarios:
    * For security testing, try to send the request without authentication or insufficient
permissions
        (expect to receive error codes like 401 or 403).
    * For performance testing, send a high volume of requests within a short timeframe
and observe
        system performance like response time and error rate.
    * Develop negative testing scenarios like providing invalid parameters, exceeding data
length,
        incorrect formats, etc. to validate API's robustness and error handling.
    * For functional testing, cover all potential combinations of optional parameters to
ensure API
        behaves as expected in different scenarios.
```

3.3.3 Test Cases Designed by SwisscomGPT Chat

We created a chat using GPT 4-Turbo model, and sent prompts to obtain test cases. AI gave 5-7 test cases per API. Each test case includes test case ID, test case description, test steps, input data and expected results. Example AI response can be found in Appendix B.

3.3.4 Test Cases Designed by Cursor Chat

In Cursor Chat, there are 6 different AI models that can be chosen. During this experiment we chose claude-3.5-sonnet. We gave Cursor Chat the same prompt generated by SwisscomGPT. Cursor Chat provided classified test cases. Each test case includes description, test data, test steps, and expected results. Example AI response can be found in Appendix B.

3.3.5 Request Accuracy

Generated test cases are as follows. To better show the difference between test cases generated by different API description accuracy, we put all contents of positive test cases in the table. To enhance readability, we only put title for all other test cases and lay test cases of same objective in the same line.

90% Accuracy	**60% Accuracy**	**30% Accuracy**

Description: Verify that the API updates the user’s address with all mandatory and optional fields correctly.
Preconditions: User must be authenticated and have permission to change the address.
Account with accountId 13004204 must exist.
Test Steps:
1. Authenticate the user and establish a secure connection.
2. Send a POST request to ”https://stesbonline01:<port>/JaxRsWeb/account/13004204/address/v2”.
3. Add a header for endUserId with value ”tgdgalu1”.
4. (Optional) Include a header for correlationId with the value ”8c38500e-2de8-11ed-a261-0242ac120002”.
5. Set the JSON Payload to:
{
”name1”: ”John Doe”,
”cityTown”: ”New York”,
”street”: ”235 E 46th St”,
”houseNo”: ”235”,
”zip”: ”10017”,
”countryCd”: ”US”
}
6. Review the request for accuracy and completeness.
7. Send the POST request to the server.
Input Data:
Headers and JSON payload as described above.
Expected Result:
The response is HTTP 200 OK.
Response JSON includes a new addressId.
	Objective: Verify that the API successfully modifies the address for a valid and existing Bill AccountID when provided with valid data.
	Prerequisites: Ensure the user is authenticated and authorized.
	Confirm the accountId exists in the system.
	Test Steps:
	1. Establish a secure connection to the server.
	2. Send a POST request to ‘https://stesbonline01:<port>/JaxRsWeb/account/accountId/address/v2‘ with the following details:
	Headers:
	endUserId: ”tgdgalu1”
	accountId: 11223344
	Request Body:
	name1: ”John Doe”
	3. Ensure the server responds with a 200 OK status.
	4. Verify the response contains a non-null ‘addressId‘.
	Expected Results: The API returns a 200 OK status and the response includes a new ‘addressId‘.
	Input Data:
	“‘json
	{
	”name1”: ”John Doe”
	}
		Objective: Verify that the API correctly changes the address for a valid Bill Account ID.
		Input Data:
		endUserId: ”tgdgalu1”
		accountId: 11223344
		JSON body:
		“‘json
		{
		”name1”: ”John Doe”,
		”street”: ”123 New Avenue”,
		”city”: ”Metropolis”,
		”zip”: ”12345”,
		”country”: ”Wonderland”
		}
		Steps:
		1. Connect to the server at ‘https://stesbonline01:<port>‘.
		2. Construct a POST request with the URL ‘https://stesbonline01:<port>/JaxRsWeb/account/11223344/address/v2‘.
		3. Add required headers:
		‘Content-Type: application/json‘
		‘endUserId: ”tgdgalu1”‘
		‘accountId: ”11223344”‘
		4. Send the prepared JSON body in the request.
		5. Validate that the response status code is 200.
		6. Confirm the response includes a new ‘addressId‘.

90% Accuracy	**60% Accuracy**	**30% Accuracy**
XSS attack prevention	No such test case provided	
Invalid data types in JSON	No such test case provided	
Unauthorized User		
Invalid AccountId		
Performance Test		
Mandatory Field Missing in JSON		30% accuracy didn't provide such test case, but a test case for SQL injection

Table 2: Test cases generated using different request accuracy

3.4 Analysis of the Results

3.4.1 Test Case Validation

To validate generated test cases, I held a meeting with senior DevOps Engineers who work on these APIs and asked their opinions about all test cases generated. We classified test cases into four categories: correct test cases, redundant test cases, non-API test cases, and error (incorrect) test cases.

3.4.2 Invalid Test Cases Overview

We set up three types of invalid test cases. In this section, I will basically describe why those test cases were determined as invalid test cases.
In esBill, users have access to all accounts. So all cross-account access here are redundant. We determined all test cases for validating the data format that will not be encountered in real world usage as redundant test cases. Test case that have overlapped functional validation with positive test case is also determined as redundant.
All performance test cases were classified as non-API test cases in our study, including test cases for load testing, response time, and concurrent access.
Test cases that evaluated non-existent functions in esBill were classified as incorrect test cases. For example, esBill doesn't have other systems connected or audit logging, so test case for cross-system verification and audit trail are wrong test cases.
This classification enables us to clearly see the quality of test cases. It not only shows the proportion of available test cases, but also focus on the common issue when using AI to generate test cases. In this way, we can analyze the performance of AI models better.

3.4.3 Test Cases Expectation

To evaluate how complete the test cases designed by AI are, we discussed and listed all the test scenarios that should be included when testing each API:
Change Address V2:
1. Basic update with only mandatory fields
2. Complete update with all fields
3. Missing accountId
4. Missing endUserId
5. Missing JSON payload or incorrect JSON format

6. Missing mandatory(name1/zip/cityTown) field in JSON
7. JSON mandatory data field exceeds the maximum length
8. JSON mandatory data field with maximum length
9. JSON mandatory data field is empty
10. Mandatory data field with special characters
11. Invalid accountId
12. Authorization test
13. Multiple updates

/ustId/v1:
1. Basic update with only mandatory fields
2. Complete update with all fields
3. Missing accountId
4. Missing endUserId
5. Missing ustId
6. UstId exceeds the maximum length
7. UstId with maximum length
8. Empty ustId
9. Invalid ustId
10. UstId with special characters
11. Invalid accountId
12. Authorization test
13. Multiple updates

Read Ust-Id: 1. Basic retrieval with only mandatory fields
2. Complete retrieval with all fields
3. Missing accountId
4. Missing endUserId
5. Retrieve Ust-Id with specific refDate
6. Invalid refDate format
7. Invalid accountId
8. Authorization test

Read Contract History: 1. Basic retrieval with only mandatory fields
2. Complete retrieval with all fields
3. Missing contractId
4. Missing endUserId
5. Invalid contractId
6. Authorization test
7. Output data structure completeness
8. Data sequence verification

3.4.4 Test Cases Designed By Different Methods

To evaluate the performance of different methods, we use metrics presicion and recall. Their calculation are as follow:

$$\text{Precision} = \frac{\text{Number of Valid Test Cases}}{\text{Total Test Case Number}}$$

$$\text{Recall} = \frac{\text{Number of Scenarios Test Cases Cover}}{\text{Expected Number of Test Scenarios}}$$

Full test case classification listings for each API are provided in Appendix C. All test cases generated by Test Case Designer(Upload) instance were valid test case. Among all test cases generated by SwisscomGPT(GPT 4-Turbo), TC6-7 of Change address v2 and TC5 of other three APIs are non-API test cases. Remainings are all valid test cases. The classification results of all test cases generated by Cursor Chat are as follows:

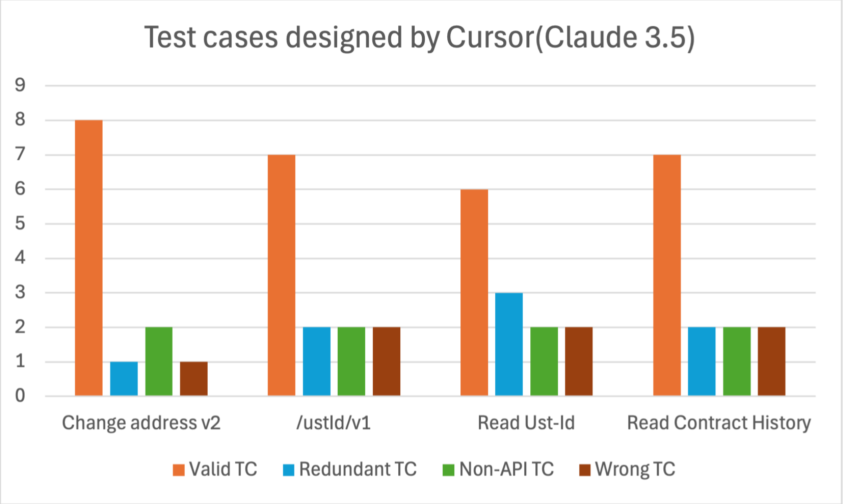


Figure 1: Test cases designed by Cursor Chat(Claude 3.5) classification

Cursor Chat considered most of the aspects that need to be tested when designing test cases, giving it a high recall. In our daily work, we did not consider about authorization and permission problems. We seldom design multiple test cases to verify data validity. Cursor Chat designed test cases for these for every APIs. Although SwisscomGPT generated only four or five test cases per API, it included test cases related to permission handling. Comparing AI-generated test cases to those created by developers, we observe that AI can produce more comprehensive test cases.

Comparing these three methods, SwisscomGPT(Upload) has a precision of 100%, which means every test cases it generated are valid. But a 26.16% recall means it missed out most of the required testing scenarios. It is only suitable when the testing errors are costly. SwisscomGPT(GPT 4-Turbo) lacked test cases that verify the validity of data formats. This is not a big issue for our APIs, as our APIs will not have a big impact on the system even if it is not fully tested. But when the test object is related to the stability and availability of the entire system, it would not be the best option to use SwisscomGPT(GPT 4-Turbo) to design test cases.

However, there is also drawback to the comprehensiveness of Cursor Chat(Claude 3.5). The precision is only 57%, which means almost half of the test cases need to be reviewed and deleted by human beings. These overcomplete and unnecessary information will consume a lot of manpower and time. I held two meetings with 3 DevOps Engineers to decide the validity of test cases. They spent more time understanding those invalid test cases than valid test cases, as invalid test cases are rare so that they required more time to fully read the texts and think. DevOps engineers thought that this overcomplete is as bad as incomplete in real software engineering.

Considering this factor, we concluded that SwisscomGPT(GPT 4-Turbo) had a better performance and could better assist our daily testing work, as it balanced between precision and recall, which means it provides enough test cases to ensure the testing objectives can be used normally, while it does not provide too many test cases, avoiding time wasting on selection and improving the efficiency. But at the same time, we could leverage the advantage of Cursor Chat when we need to test important APIs that need to be fully tested. It is suitable for exploratory testing and early of testing to discover potential issues. A better strategy is to ask Cursor Chat to provide points that need to be tested instead of complete test cases at first. After checking test points required, we can then ask Cursor Chat to provide detailed test cases including input and test steps.

	SwisscomGPT (Upload)	SwisscomGPT (GPT 4-Turbo)	Cursor Chat (Claude 3.5)
Total Number of TC	16	22	51
Precision	100%	77.27%	54.90%
Recall	26.19%	38.10%	73.81%

Table 3: Test cases designed by different methods

3.4.5 Test Cases Designed Using Different Request Accuracy

We can see from the answer from AI that test cases based on 90% accuracy contains detailed information, such as fields in JSON payload, correlationId and specific accountId. Test cases based on 60% accuracy didn't contain such detailed information, but it also had clear structure and contained information that is detailed and clear enough for testing. However, it lacked enough boundary value testing and had fewer testing scenarios. Test cases based on 30% accuracy covered basic functions, but it didn't organize in a neat structure. For example, it did not provide preconditions and expected results. Additionally, it had fewer testing scenarios, and the logic of some test scenarios is not complicated enough. For instance, the JSON payload in SQL injection case is:

```
{
”name1”: ”’; DROP TABLE Address;–”,
”street”: ”hacked’ OR ’1’=’1”
// ...other required fields
}
```

But this does not mean the best test cases should be designed based on 90% accuracy and we can't use test cases designed based on 30% accuracy. Instead, in real-world software engineering, we also need to consider time and human resource under this case. Writing highly detailed instructions requires significant time. In my experiment, we spent more than one week to restructure the API descriptions based on original Wiki page, find suitable test data and apply instructions inside prompt. This process took more time than simply providing a prompt and generating test cases using AI. When testing a more complex object, requirements for implementation costs and testing resource can be very high.

Based on all considerations above, we suggested to use 90% accuracy test cases if the tested objects are very crucial that need to be fully evaluated regarding their functions, error handling, security and performance. 60% accuracy test cases contains enough information for most test requirements. If tested objects just need a quick and basic functional check, then 30% accuracy test cases is enough, but there is a risk that you will miss out critical issues.

4 AI for Automated Test Code Generation

4.1 AI integration in Test Code Generation Overview

There are a lot of other AI tools that have been developed to assist with code generation and have been revealed in other studies. FreqCCS, ArCCS, and BMNCCS[11] are systems that can complete code by learning existing code repositories. Study from Barke et al. used grounded theory to reveal how programmers interact with code generation tool, Github Copilot[12]. Randoop[13] is a unit test code generation tool for Java which uses feedback-guided random test generation. TestSpark[14] is a plugin for IntelliJ IDEA that enables users to generate unit tests with a few clicks by creating a feedback cycle between the IDE and the LLM. Raychev et al.[15] simplified the code completion problem into a natural language processing problem to predict the probabilities of sentences and managed to solve the code completion problem. Although my study does not directly utilize these tools, they lay a solid foundation for AI-assisted test code generation and give me an overview and understanding in this area.

Today, some popular AI code generation tools that we took into consideration include Copilot, Cursor, Tabnine, and Amazon CodeWhisperer. Tabnine's understanding of code is limited compared to more advanced models like GPT-4o[16], which have broader context awareness. Amazon CodeWhisperer is mainly for AWS ecosystems. Copilot is one of the most popular AI tools, as it has OpenAI models and can be used in various IDE such as VSCode as plug-in. But to use Copilot you need a paid subscription. Cursor also need to pay if you want to try its full functions, but there are some basic AI models that can be used freely. Meanwhile, we can chat with AI when using Cursor and it has its own IDE. Since we can use advanced models and chat freely in Cursor and also it has a familiar IDE, we used Cursor in our study.

4.2 Methodology

4.2.1 Exploration of Cursor Capabilities

In Cursor, there are three AI functions: Chat, Composer and Bug Finder. We used Chat during test case design section, which is quite the same as other AI Chat. In this section, we would use Composer, which can help us to generate code directly in the project.

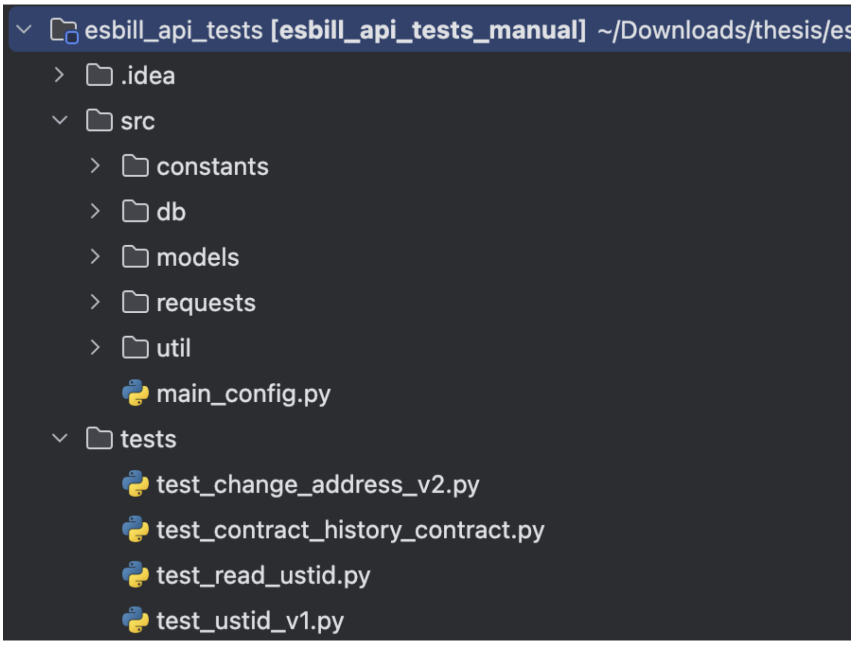
There are 6 AI models available in Cursor Composer, which are Claude-3.5-sonnet, Claude-3.7-sonnet, Claude-3.7-sonnet-thinking, GPT-4o, GPT-4o-mini and o1. However, the high demand making Claude-3.7 often unavailable, and we can't use o1 if we are not using Cursor Pro. Due to these constraints, we chose Claude-3.5-sonnet and GPT-4o for our experiments.

Claude-3.5-sonnet is developed by Anthropic. It has great support to developing language such as Python, Java, and SQL. It is outperformed in test code generation and context understanding. GPT-4o is developed by OpenAI. It is cheaper and faster than GPT-4o-Turbo, and

supports multimedia input.
In Cursor Composer, we can select specific documents in project to AI when using Composer. Since the test code will be generated in our existing testing project which already has a clear project structure and test code for other APIs, we would utilize this function to let AI better understand the project and organize generated code.

4.2.2 Project Structure

The existing code analysis ability of Cursor can analyze the currently visible files and distinguish structures such as functions, classes, and modules. It does not require human training to understand the project structure. But we can specify the files needed when using Cursor Composer to let AI focus on those files. In this chapter we are going to briefly introduce our project structure to have a clearer overview to the project.
Before the experiment, the project only contained test code for webhook APIs that are used to invoke functions in another system, which means that these test code follow a different calling mechanism from our APIs. In this case, AI should not generate test code by simply modifying parameters in the existing test code, but it needs to understand and generate a different code structure to properly invoke our APIs.
The testing project uses a pytest framework and PostgresSQL as database. As shown in Figure 2, the project contains two subdirectories, which are /src and /tests. /tests includes python scripts that invoke test functions and assert results. /src contains all other elements needed for the testing, including database configuration under /db, test data under /constants, request functions under /request, and other helper methods. Authentications and URLs are stored in main_config.py under /src. Since main_config.py contains privacy data that give access to both the EsBill system and the database, we faked authentication data in this file when we opened the project in Cursor and used real data to test our code in IntelliJ IDEA.



```
esbill_api_tests [esbill_api_tests_manual] ~/Downloads/thesis/es
  .idea
  src
    constants
    db
    models
    requests
    util
    main_config.py
  tests
    test_change_address_v2.py
    test_contract_history_contract.py
    test_read_ustid.py
    test_ustid_v1.py
```

Figure 2: Project Structure

4.2.3 Design of Test Code Generation Prompts

We have generated test cases for these APIs in the previous section. We would use the same four APIs for test code generation, and prompts used here section were modified based on test cases from the last section. For consistency, the same initial prompts were used for both AI models. Subsequent modifications were recorded and analyzed separately.
To ensure reusability of test cases and modularity, we organized prompt by separating it to task introduction, API description, database verification and test case part. In this way it would be easier for developers to understand and reuse this prompt in the future. Additionally, unified the format for API testing and ensured each test case follows the same structure to enhance consistency of prompt. Following is the final version of prompt for Change Address V2:

> Read these files and understand how to write API tests and connect the database in this project. Then write code based on the following test case for the API change address v2. Set verify=False in the request. Don't put all codes in one file, but put test script in /tests, test data in /src/constants, request functions in /src/requests, database configuration in /src/db/db_config.py , access and URL in /src/main_config.py, headers and json transformation in src/util/util.py

Task introduction and code organization requirement Here, we describe our task to AI. We selected existing code files(/tests/test_customer_replication.py, /src/request/customer_replication_req.py, /src/db/db_config.py, /src/constants/test_addresses.py, /src/util/util.py, and /src/main_config.py) that show how the existing test code is organized inside the project and asked AI to learn how to organize the test code. To avoid SSL connection error, we specified "Set verify=False in request" in the prompt. As AI tended to put all test code inside one Python script, we described the contents inside different files and asked it to put code separately at the end of this paragraph.

> URL: https://stesbonline01.corproot.net:26443/JaxRsWeb/account/:accountId/address/v2
> (please form it using the variants in main_config.py)
> HTTP method: POST
> Authentication:
> {
> ESB_ACCOUNT: "esonline",
> ESB_ACCOUNT_PSW: "esonline"
> }

API description Here we give information of API to AI. We stress again to put URL in main_config.py. The authentication data here are fake data.

> For test cases with status code 200, add a database assertion to verify that all fields are updated in database. Table constants.ENV.tadsupp has timestamp, address_id, location_id, name1, name2, first_name(firstName), name_addr_suppl(nameSupplement), co_remark (coRemark), po_box_no(postfach). Table constants.ENV.talocda has location_id, country_id, zip, city_town(cityTown), street, house_no(houseNo), location_suppl (locationSupplement), zip6, residence_descr(residenceDescr). Table constants.ENV.tadrusg has address_id, usage_integer1, user_id. Table tacodat has country_id and country_short_iso (countryCd). Our data is the latest record by filtering user_id and usage_integer1. The user_id should match endUserId, and usage_integer1 should match accountId. When coun-

tryCd in test data is None, country_short_iso in database will be set as 'CH'. location_suppl in database is always None. po_box_no in database only takes postfach in the test data from the 9th letters. Please write a correct database verification code based on this information.

Database Verification EsBill system is connected to a Postgres database. All information updated to the system will be updated in the database at the same time. After discussing with colleagues, we included a database test in our test code to check that the POST APIs were fully working. The project had already included the database in constants.ENV variable, so we told AI which tables contain the relevant fields, which fields need to be checked, and how to validate them.

TC1: Basic Address Change
Description: Verify successful address update with mandatory fields
Test Data:
AccountId: 13004204
Headers:
{
Content-Type: "application/json",
endUserId: "TAAGUXI1"
}
- Payload:
{
"name1": "John Doe",
"zip": "3011,
"cityTown": "Bern"
}
Test Steps:
Set up POST request with headers
Add mandatory payload fields
Send request
Verify response
Expected Results:
Status code: 200
All fields updated correctly in database
TC2: Complete Address Update
...

Test case Here are valid test cases selected from test case design section. We edited expected results to let AI to check to the API.

The prompts for other APIs follow the same structure, but GET APIs did not contain database check, since they did not update information in the system. All test cases we used are as follow:

Test Case ID	Change address v2	/ustId/v1	Read Ust-Id	Read Contract History
TC1	Basic Address Change	Basic UstId Update	Basic Ust-Id Retrieval	Basic Contract History Retrieval
TC2	Complete Address Update	Field Length Validation	Invalid Account ID	Invalid Contract ID
TC3	Missing Mandatory Fields	Invalid Account ID	Unauthorized Access	Unauthorized Access
TC4	Invalid Account ID	Missing UstId	Inactive account ID	Multiple Contract Data Completeness
TC5	Field Length Validation	Missing Account ID	-	-
TC6	Special Characters	Unauthorized Access	-	-
TC7	Authorization Test	Inactive account ID	-	-
TC8	Inactive Account ID	-	-	-

Table 4: Test cases used to generate test code

4.2.4 Validation of Generated Test Code

Before asking AI to generate the test code, we first wrote all test code myself and recorded the time spent. In this way, we found errors that needed to be told to AI in advance(such as set verify=False) and was able to compare the efficiency of AI and human beings.

After generating test code using AI, we first checked if all test cases had been included and if there were basic code organization problems. If these situations occurred, we would directly tell AI the problem and solution of its code and ask AI to generate the test code again without executing the code.

To verify the validity of the test code, we used the command "pytest" to execute all generated test code in IntelliJ IDEA. If serious errors show up, we would ask AI to correct the code again. If the errors were small and can be corrected in minutes, we would correct the bugs myself. We regenerated and corrected the code multiple times until all test cases passed and recorded the total time spent.

During the eight verifications, only test code for Read UstId generated by Claude-3.5-Sonnet passed after the first generation. We had to deal with bugs or improve our prompts in most cases. Test code generated by AI had many big and small errors, that need to be run and modified repeatedly before the code can be run correctly. We needed to check not only code logic and grammar but also code organization and execution information. This is a cumbersome process and cost us about half of our AI experiment time.

4.3 Experiments

4.3.1 Manual generation

As mentioned before, we first wrote the test code manually. The first API we wrote the test code is Change Address V2. We spent most time on the first test case to deal with SSL verification problem (which is what set verify=False in the prompt deals with) and database field checking. In default, requests library will check the validity of the SSL certificate. If we didn't set verify=False, test request will fail directly which is not the problem of code logic, and the experiment could not continue. Since the error did not give us a clear instruction to set verify=False, we spent a lot of time searching solutions for this issue on the Internet. Disabling SSL verification in a production environment is risky, but in the test environment this is acceptable. These two parts cost 5 hours. In retrospective, the SSL verification issue can be handled more efficiently by controlling the verify option using environment variables or let all APIs reuse the same session, so we do not need to set it separately in every request. After figuring out these problems, the remaining test cases were quite simple and cost only 1 hour in total.
The test code for other APIs were also easy after writing code for Change Address V2. Read UstId cost 1 hour to form the frame of testing GET API, and Read Contract History only cost 0.5 hour. /ustId/v1 cost 2 hours as it is also a POST API that requires database checking.

4.3.2 Code generated by GPT-4o

Cursor generated code with a brief explanation and step-by-step breakdown which shows files and part of codes it modified. Below is an example of Cursor's reply:

> I'll help create the test structure for the UstId retrieval API. I'll break this down into multiple files.First, let's update the main configuration:
> // code modified in main_config.py
> Let's create test data for UstId:
> // code modified in test_ustid_data.py
> Let's update the util.py to include headers for UstId:
> // code modified in util.py
> Let's create the request handler for UstId:
> // code modified in ustid_req.py
> Finally, let's create the test file:
> // code modified in test_ustid.py
> This implementation:
> Creates a dedicated test data file for UstId tests
> Adds a new header utility function for UstId requests
> Creates a request handler that:
> Measures response time
> Handles authentication
> Provides proper error handling
> Creates test cases that:
> Verify successful retrieval with response time check
> Validate response format and UstId format
> Test error cases (invalid account, unauthorized, inactive account)

Use consistent error handling across all tests

The code follows the project's patterns and reuses existing infrastructure while adding UstId-specific functionality.

Let me know if you need any clarification or have questions about specific parts of the implementation!

As mentioned before, we asked AI to correct bugs inside the code multiple times until the code only contained flaws that we could correct manually. The prompts used, the bugs identified in the generated code, and the corresponding corrections are as follow: **/UstId/v1**

Item	Prompt	Bugs
AI's 1st generation	Initial prompt with API description and test cases	AI set accountid/ustid = ” in functions test_missing_ustid()/test_missing_account_id() instead of accountid/ustid = None
Manual work	Set accountid/ustid = None	
Time spent	Prompt preparation & file selection 15min Code generation 5min Debug & execution 5min	

Table 5: GPT-4o code generation for /UstId/v1

Change Address V2

Item	Prompt	Bugs
AI's 1st generation	Initial prompt with API description and test cases	AI put all test code in one file (test_address_change.py)
AI's 2nd generation	Don't put all codes in one file, but put test code in /tests, test data in /src/constants, request functions in /src/requests, database configuration in /src/db/db_config.py, access and URL in /src/main_config.py, headers and json transformation in src/util/util.py	Separated most test code, but didn't clean up unused database function in test_address_change.py
AI's 3rd generation	Please put test data in /src/constants, and clean up unused functions in test_address_change.py	Put test data in the right file, but didn't clean up unused database function in test_address_change.py Test case test_inactive_account_id didn't use request function in address_change_req.py but rewrote another request function
Manual work	1. Deleted redundant function verify_database in test_address_change.py. 2. Corrected country_id to country_short_iso in database verification part. 3. Modified test_inactive_account_id to use request function in address_change_req.py.	
Time spent	Prompt preparation & file selection 30min Code generation 25min Debug & execution 35min	

Table 6: GPT-4o code generation for Change Address V2

Read Contract History

Item	Prompt	Bugs
AI’s 1st generation	Initial prompt with API description and test cases	Didn’t generate code for test case Contract Data Completeness but generated test data for this test case
AI’s 2nd generation	You didn’t provide test code for Contract Data Completeness	Provided test code for Contract data completeness, but had errors in response assertion: it asserted ‘status’ and ’details’ which were not exist in the response and asserted contractId is string which is in fact integer

AI’s 3rd generation	[
	{
	”contractId”: 6096298,
	”foreignContractId”: ”6096298”,
	”accountId”: 8413466,
	”scn”: 1439,
	”validFrom”: ”2024-08-01”,
	”validTo”: ”9999-12-31”,
	”userId”: ”DKTMKEC5”,
	”timestamp”: ”2024-08-30 08:59:00.917000000”
	},
	{
	”contractId”: 6096298,
	”foreignContractId”: ”6096298”,
	”scn”: 1439,
	”validFrom”: ”2013-03-01”,
	”validTo”: ”2024-07-31”,
	”userId”: ”DHUZKW35 “,
	”timestamp”: ”2024-08-30 08:59:00.896000000”
	}
	]
	This is the response when status code= 200. Could you add fields and data form check for test case with status code = 200?
		Corrected fields and data form checking for test function test_contract_data_completeness(), but didn’t add for test function test_basic_contract_history_retrieval()

AI’s 4th generation	Please also add these checks to test_basic_contract_history_retrieval	Added same code to test_basic_contract_history_retrieval()

Item	Prompt	Bugs
AI's 5th generation	These codes seem to be redundant, could you put the fields and form check inside one function?	Put fields and form check inside one function and called the function in both test cases. However, there is no accountId in second history record if one contractId contains multiple contract history records, but AI verified for every record, which caused test to fail
Manual work	Deleted the logic of checking accountId in every records	
Time spent	Prompt preparation & file selection 10min Code generation 5min Debug & execution 25min	

Table 7: GPT-4o code generation for Read Contract History

Read Ust-Id

Item	Prompt	Bugs
AI's 1st generation	Initial prompt with API description and test cases	AI asserted if response.json() contains 'error' for negative test case, which is redundant since we checked HTTP status code to check status AI put authorization in function test_unauthorized_access() instead of main_config.py
AI's 2nd generation	You didn't delete the authorization in test_unauthorized_access()	AI corrected it and test passed
Time spent	Prompt preparation & file selection 15min Code generation 5min Debug & execution 5min	

Table 8: GPT-4o code generation for Read UstId

From the execution results, we can see that GPT-4o actually improves the efficiency for test code generation, and it gave quite clear descriptions and hints to the generated code. GET APIs had very few errors after the first generation. It had code organization problem at the first generation of the first API, but it immediately corrected it at the next generation. But there was a lot of redundant code during generation, which suggests that we need to carefully check how the generated code organized to improve code quality even when there is no error inside the code. Notably, those errors caused by incomplete API or test case information indicated us that the prompts still need to be checked and include more information. Overall, GPT-4o matches our expectation that it reduced the workload of developers.

4.3.3 Code generated by Claude-3.5-Sonnet

When experimented with Claude-3.5-Sonnet, we improved our prompt based on the experimental results of GPT-4o, and we spent less time on the experimental process, as we were already familiar with the process. However, the Claude-3.5-Sonnet experiment was not as efficient as we expected, as the code it generated had a lot of logic error. We will further compare the performance of two AI models in the next section. The prompts used, the bugs identified in the generated code, and the corresponding corrections are as follow:

Read UstId

Item	Prompt	Bugs
AI's 1st generation	Initial prompt with API description and test cases	Test passed
Time spent	Prompt preparation & file selection 5min Code generation & check 5min Debug & execution 5min	

Table 9: Claude-3.5-Sonnet code generation for Read UstId

Read Contract History

Item	Prompt	Bugs
AI's 1st generation	Initial prompt with API description and test cases	Returned imestamp from database has 9 digits' nanosecond but datetime.strptime() can only fit in 6 digits' nanosecond
Manual work	Deleted last 3 digits in nanosecond for assertion	
Time spent	Prompt preparation & file selection 5min Code generation & check 5min Debug & execution 15min	

Table 10: Claude-3.5-Sonnet code generation for Read Contract History

Change Address V2

Item	Prompt	Bugs
AI's 1st generation	Initial prompt with API description and test cases	Used ODBC instead of PostgresDB Didn't provide test case without authorization Put test data in wrong test data file
AI's 2nd generation	Please: 1. use psycopg2 instead of pyodbc in db_config.py 2. provide the code for test_authorization 3. put test data in other place instead of test_customers.py, as it's not customer data	Fixed all bugs mentioned, but address data still in wrong file

Item	Prompt	Bugs
AI’s 3rd generation	Also put TestAddressData in other place instead of test_customers.py, as it’s not customer data	Moved TestAddressData to right file, but didn’t generate code for function get_headers_with_user() in util.py it used
AI’s 4th generation	You have error at headers = util. get_headers_with_user (customer[”user_id”]) in address_change_v2_req.py. Please check	Added util.get_headers_with_user() and logs in address_change_v2_req.py. Remaining bugs: 1. Changed import sentence ’from src import constants’ to ’from src. constants import constants’ which is wrong 2. Used ? in query. In postgres it should be %s 3. Didn’t use fetch_one() or fetch_all() in the query, so the query always returned None 4. Results from db is dict, but was asserted as a list
AI’s 5th generation	The query result is dict but your assertion assumes it’s a list, please correct your code	1. Prompt: “take postfach from the 9th letter in test data, while code is: assert db_data[’po_box_no’] == expected_data[”postfach”][8:] 2. Wrong status assertion: exc_info. value.status_code is a string error message but it was handled as a int
AI’s 6th generation	exc_info.value.status_code is a string with the form as ”AssertionError (’PUT request failed:\n Status code: 400\n Error: 400 Client Error: Bad Request for url: https:// stesbonline01.corproot.net:26443/ JaxRsWeb/account/13004204/ address/v2’)”. Please correct your code of verifying all test cases with status code != 200	1. Put a invalid user credentials in test case “without authorization”(set user_id = ”INVALID_USER”) which is wrong 2. with pytest.raises(HTTPError) as exc_info: can’t catch error message correctly
Manual work	1. Corrected import sentence to ’from src import constants’. 2. Corrected ’?’ in database query to ’%s’. 3. Added fetch_one() in database query. 4. Changed with pytest.raises(HTTPError) as exc_info to raises(Exception)	
Time spent	Prompt preparation & file selection 5min Code generation & check 60min Debug & execution 40min	

Table 11: Claude-3.5-Sonnet code generation for Change Address V2

/UstId/v1

Item	Prompt	Bugs
AI's 1st generation	Initial prompt with API description and test cases	Didn't provide standard UstId test data
Manual work	Set UstId test data	
Time spent	Prompt preparation 5min Code generation & check 7min Debug & execution 8min	

Table 12: Claude-3.5-Sonnet code generation for /UstId/v1

4.4 Analysis of the Results

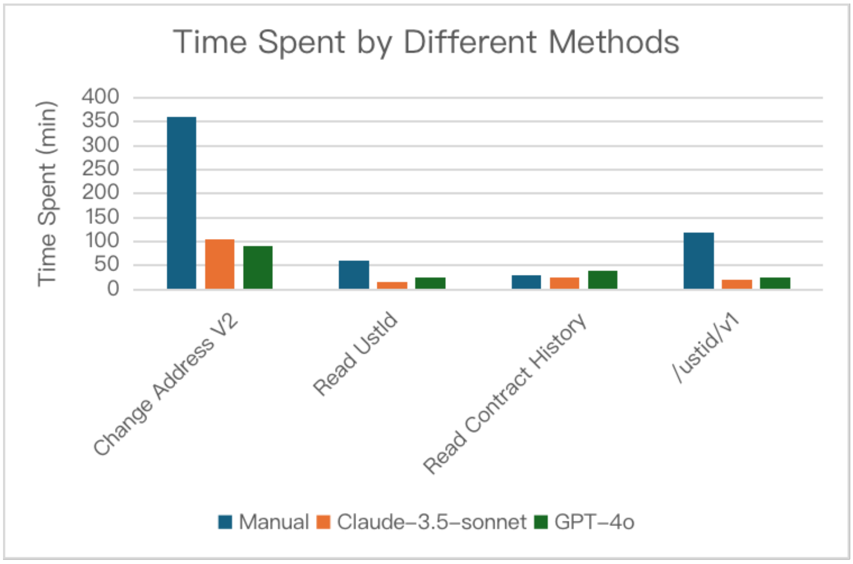


Figure 3: Test cases designed by Cursor Chat(Claude 3.5) classification

Comparing time spent by AI models and generating code manually, we can observe that AI is more efficient than humans, particularly when handling POST APIs. While existing code quality issues, such as logic errors and code redundancies required manual debugging, leading to a trade-off between speed and accuracy, AI still provided an overall efficiency compared to manual coding.

However, manually generating code took almost the same time as using AI when dealing with Read Contract History. The reason why manual work and AI cost similar amount of time is due to the simplicity of GET API test. Unlike complex APIs such as POST APIs, GET APIs require fewer test cases and do not require database verification, which significantly reduce the effort for developers when writing test code. However, when using AI, it requires additional time on prompt generation, code checking, and output verification. This offset its potential time savings of fast code generation. Considering about this, we would not recommend using AI if you only need to test very few GET APIs.

To better analyze the bugs that occurred when using AIs and compare these AI models, we divided the bugs into six different categories. **Configuration Error** are errors related to incorrect configurations. **Test Case Omission** represents the situation in which the test cases were provided on the prompt but AI did not generate code for these test cases. **Code Redundancy** refer to unnecessary or duplicate code in the project. **Code Organization Issue** shows code placed in the wrong location that did not match the project structure. **Logic Error** are errors with the code's logic. **Test Case Design Flaw** are errors that occurred due to the imperfect description of the prompt.

Bug Type	GPT-4o	Claude-3.5-sonnet
Logic Error	1. Mistook country_short_iso and country_id 2. Put authorization in test_unauthorized_access() 3. Didn't add data form check for test_basic_contract_history_retrieval() 4. Set accountid/ustid = ‘’ in test_missing_ustid and test_missing_account_id instead of None	1. No get_headers_with_user() in util.py as it used 2. from src import constants is wrong 3. Used ’?’ in query instead of ’%s’ 4. No fetch_one() or fetch_all() in query 5. Results from database were dict but were asserted as list 6. Prompt: ”takes postfach in test data from the 9th letters”; code: assert db_data[’po_box_no’] == expected_data[”postfach”][8:] 7. Wrong status code assertion: exc_info.value.status_code is a string error message but it was treated as a int 8. Put wrong invalid user credentials in ”without authorization” 9. Used with pytest.raises (HTTPError) as exc_info that can't catch error message correctly
Code Redundancy	1. didn't clean up unused database function in test address change.py 2. test_inactive_account_id() didn't use request function in address_change_req.py but rewrite function 3. Asserted ’error’ in response.json() after HTTP status code check 4. Same code in test_basic_contract_history_retrieval() and test_contract_data_completeness()	-

Bug Type	**GPT-4o**	**Claude-3.5-sonnet**
Code Organization Issue	1. AI put all test code in one file (test address change.py)	1. Put test data in wrong test data file 2. Address test data still in wrong file
Test Case Design Flaw	1. Asserted 'status' and 'details' which were not exist in the response and asserted contractId is string which is in fact integer 2. No accountId in second history of same contractId, but AI verify for every history	1. datetime.strptime(timestamp, "%Y-%m-%d %H:%M:%S.%f") can only assert 6 digits nanoseconds but returned timestamp has 9 digits 2. Didn't provide standard UstId
Configuration Error	-	1. Used ODBC instead of PostgresDB
Test Case Omission	1. Missed multiple contract history test case	1. Didn't provide no authorization test case
Total	**12**	**15**

Table 13: Bugs analysis during test code generation

We can see that the code generated by GPT-4o has fewer errors than the code generated by Claude-3.5-sonnet. We tested Claude-3.5-sonnet later than GPT-4o and were more familiar with the process and had better prompt at that time, but we did not feel more efficient. Instead, we felt that we spent a lot of time dealing with basic grammar mistakes. In Cursor, the default model is Claude-3.5-sonnet. We highly recommend to change the model before using it.

Another issue worth noting is test case design flaw errors exist with code generated by both models. These errors were mainly caused by our carelessness and could theoretically be avoided. These prompts were modified based on the test cases generated by AI. We mentioned the over-complete problem of test cases generated by AI in the previous chapter, and these prompts were iteratively optimized according to discovered errors during the process of code generation. We can say that the information in the prompts was very complete, but test case design flaw errors still exist. When generating a prompt in a real project, we should pay attention to not only the test objectives but also the data format of different data during the whole process, including input data, output data, and even intermediate data. But it is very hard for humans to cover all these details at first. Many details need to be discovered during the process of code generation and testing. We think that using AI for test code generation can improve efficiency compared to manually generating code, but it still requires a lot of human work nowadays.

Our findings indicate that AI significantly improves the efficiency of test code generation, especially for complex testing objectives such as POST APIs. Although GPT-4o has better performance with fewer logic errors, both GPT-4o and Claude-3.5-sonnet require careful human work for prompt generation and correctness verification. We conclude that AI is a valuable assistant for test generation, but is not yet a fully automatic tool. For further research, it is valuable to let third party experts evaluate test code generated by human and AIs without letting them know code author to provide more convincing qualitative assessments.

5 Broader Discussion

In this section, we are going to discuss the application of AI tools like SwisscomGPT and Cursor in real-world software testing, and concerns such as security and data handling when using AI.

5.1 Interplay Between Human and AI Contributions

From our study, we can see that AI has some advantages over developers in software testing. For example, when generating test cases, AI tends to cover all exceptions and edge cases, which can give developers more design inspiration. When it comes to test code generation, AI has great efficiency if a large number of similar and cumbersome business codes are needed. It can analyze and understand code in large projects at high speed.

However, AI still has shortcomings. We still need human developers to analyze business logic and user scenarios and handle data that AI cannot understand and analyze. AI output also requires human quality control, such as test case selection and optimization, and test code optimization and bug fixes.

Considering about this, future research for AI-assisted software testing can focus on reinforcement learning with human feedback, to let AI study the optimization strategies of human beings. By introducing human optimization strategies, AI should improve its ability to optimize the selection of test cases, the detection and correction of bugs, the adjustment of the test strategy based on business requirements, and adaptive learning to reduce the cost of testing time.

5.2 Privacy and Security Concerns in AI-assisted Testing

We didn't need to deal with data privacy in this study, since all data outside Switzerland are fake data. But most of the time, when using AI to assist in software testing, a great amount of sensitive data is involved, especially in the government, medical care, and finance areas. There are potential risks in exposing sensitive data. The introduction of privacy regulations such as GDPR also makes it challenging to ensure that AI systems comply with these regulations.

To improve these issues, we should use advanced encryption techniques to protect data during storage and transmission. Data anonymization and de-identification techniques still need to be further improved. To let users clearly understand how the data is handled, everyone should engage in improving the interpretability of AI models and strengthening supervision and audit of AI. It will be better if AI is deployed on-premises, but not in the cloud.

5.3 Handling Multilingual and Outdated Data

EsBill team faces these issues, and this situation is widespread throughout the world. In the era of globalization, more and more companies have to handle multilingual data. However, existing AI models are mainly trained on English datasets, and support data in other languages not as well as English. To solve this issue, we should introduce more multilingual data to AI training data and use multilingual pre-train models such as mBERT and XLM-R to train AI. AI should also be specially trained combining local culture background.

Meanwhile, software products, especially those in large and stable companies, have gone through decades of iteration. This involves a lot of outdated data. On the other hand, in today's fast-developing society, information is constantly updated, and AI also has the problem of outdated training data. Retraining a large AI model is very costly and companies may not update training data in time. Considering about this, AI knowledge base needs to be updated regularly, and use dynamic training methods such as online learning to train AI in real-time streaming. Combining internet search and utilize user feedback can also improve this problem.

5.4 Specialized AI Models for Niche Applications

Specialized AI models are playing an important role in several industries such as medical diagnosis, legal analysis, financial risk assessment, etc. However, these industries involve relatively niche markets, which leads to problems such as insufficient data sources, small capital investment, and strong task-orientation. AI models have high development and maintenance costs and weak cross-domain migration capabilities.

With the rapid development of AI infrastructure, we will have more powerful cloud computing and automated machine learning capabilities that reduce the deployment cost of AI in niche applications. Meanwhile, modular AI components also contribute to this issue.

5.5 Integration of AI Tools into Industry Settings

Nowadays AI tools are developing rapidly. While I was writing this paper, a new AI from China, DeepSeek, came out. It achieved the performance of top AI models such as GPT-4, Claude 3, and Gemini 1.5 with significantly lower training costs, and is open source, shocking the whole world.

Even though we have seen that AI can improve the efficiency of software testing, industry still faces challenges besides all concerns discussed above. For example, how to compatible with existing CI/CD pipelines and how to seamlessly integrate with DevOps processes, trade-off between computing power and cost, and issues regarding industry standard setting.

AI should be designed to be more modular and flexible. AI platform can support DevOps tools and technology stacks, and provide plugins and API interfaced for popular CI/CD platforms such as Jenkins and GitLab. AI tools can balance computing power and cost by load balancing and cloud services. Meanwhile, AI companies can collaborate with industry standardization associations to push industry standards setting forward.

5.6 Potential Risks of AI-generated Testing Beyond Technical Aspects

Current research on AI-assisted testing focus more on technology side, such as coverage and accuracy. However, with further deployment of AI in the testing area, the potential risks in organization and legality are worth to discuss.

From organizational aspect, AI-assisted testing may weaken the skill accumulation of test engineers. For example, test engineers may rely more on prompt engineering instead of test case design itself, which will affect the work division and capacity building of the team.

From legality aspect, there is no clear legal framework to clarify the responsible person for errors caused by AI. For example, if the code generated by AI causes huge losses, should the AI developer, or the AI user and operator be responsible for this issue?

Therefore, future research should not only explore the technological improvement, but also legal compliance, human-machine collaboration ethics, and organizational structure adjustment.

5.7 Generalization Beyond the EsBill Project

Although the study is based on the EsBill project, the methodologies and findings still take implication in other projects. The AI application module have great inspiration on other similar enterprise project, especially those with a long development history, have knowledge of different languages on different platforms. For example, how to leverage LLM model to retrieve knowledge from legacy documents, design test cases and generate test codes.

In addition, the study about the analysis to performance of different AI models, provides a reference for enterprises when choosing AI tools as assistance. The study indicates that the usability and accuracy of AI-generated content is limited by the structure of context. Therefore, when promoting the methodologies in other scenarios, it is necessary to make personalized adjustments according to specific systems. For example, modern web systems based on microservice architecture contain a lot of API interfaces, which is suitable for the test case generation method proposed in this study. However, it needs to integrate the CI/CD process to ensure the test code can adapt to quick interface changes. In traditional ERP systems or banking systems, interfaces are stable but projects are highly dependent on legacy documents. These systems are suitable for the knowledge retrieval techniques proposed in this study.

6 Conclusion

The study suggests that, for documenting software testing, AI performs better on structured Wiki text but cannot process tables, images, or database DDL files. In the test case design, Cursor Chat is good at exploring test scenarios, but SwisscomGPT is good at providing essential test cases. The accuracy of the request has a great impact on the generated test cases. In terms of test code generation, AI can quickly understand the project framework and generate a test code draft, but human work is still required for prompt generation and correctness verification.

Based on the experimental findings, the advantages of AI are significant time savings, standardized output, and suitability for first draft generation. However, AI has difficulty processing knowledge of specific structures, such as complex database DDL files and tables. The generated test cases still need to be manually screened, and the correctness of the generated test code cannot be guaranteed. Although this study is based on EsBill project, the experimental results still provide inspiration on how to integrate AI into the software testing process. Therefore, we propose some practical suggestions for integrating AI into the software testing pipeline.

1. Pre-process knowledge with data types that AI cannot process into a more accessible format to ensure AI can effectively process and generate relevant information.
2. Choose AI tools developed by the enterprise for the extraction of information from internal documents.
3. When designing test case, leverage general AI to explore potential test scenarios for critical APIs or complex systems, and then use enterprise AI to generate structured and

clear test cases.

4. The accuracy of the request should be selected based on the criticality of the system. For critical systems, we recommend writing prompt with 90% description accuracy. For less critical systems, 60% accuracy is sufficient for most tests. For basic functional checks, 30% accuracy can be considered.

5. AI should be used to speed up the process and generate first drafts, but manual verification and refinement are crucial to ensure the generated test code is executable and meets the required standards. GPT-4o performs better on this task than Claude-3.5-sonnet.

6. For each stage of AI participation, human review of the generated results is required before they are applied to the project.

Although we conducted experiments on the entire testing process, this study still has limitations. This study is based on backend API testing and does not cover areas such as GUI, mobile, and embedded systems. Moreover, the study is based on the specific context of EsBill project. Its conclusions and scope of application are still limited and cannot be fully generalized to all types of AI-assisted software testing.

Based on the findings and improvements made during this study, we suggest several key areas to explore in future research on AI-assisted software testing. For AI tools, reinforcement learning with human feedback, modularization of AI systems and their integration into CI/CD platforms and DevOps tools can greatly improve the overall efficiency of AI-assisted testing systems. In addition, how to use cloud services to balance computing power and cost is worth studying. Moreover, future work should explore AI-assisted testing in different fields, including GUI, mobile, and embedded systems to expand the generalizability of AI testing tools. At the same time, encryption and data anonymization technologies need to be continuously developed to address data security and privacy issues. By integrating these improvements, AI can reduce the cost and complexity of software testing, making it a more powerful tool for a wider range of applications.

References

[1] Aakash Ahmad, Muhammad Waseem, Peng Liang, Mahdi Fahmideh, Mst Shamima Aktar, and Tommi Mikkonen. Towards human-bot collaborative software architecting with ChatGPT. In *Proceedings of the 27th international conference on evaluation and assessment in software engineering*, pages 279–285, 2023.

[2] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code. *arXiv preprint arXiv:2206.01335*, 2022.

[3] Youyang Ng, Daisuke Miyashita, Yasuto Hoshi, Yasuhiro Morioka, Osamu Torii, Tomoya Kodama, and Jun Deguchi. SimplyRetrieve: A private and lightweight retrieval-centric generative AI tool. *arXiv preprint arXiv:2308.03983*, 2023.

[4] Sara Bouzid and Loïs Piron. Leveraging generative AI in short document indexing. *Electronics*, 13(17):3563, 2024.

[5] Marco Rospocher, Francesco Corcoglioniti, and Mauro Dragoni. Boosting document retrieval with knowledge extraction and linked data. *Semantic Web*, 10(4):753–778, 2019.

[6] Simon Knollmeyer, Muhammad Uzair Akmal, Leonid Koval, Saara Asif, Selvine G. Mathias, and Daniel Gromann. Document knowledge graph to enhance question answering with retrieval augmented generation. *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2024.

[7] Simon Knollmeyer, Muhammad Uzair Akmal, Leonid Koval, Saara Asif, Selvine G Mathias, and Daniel Großmann. Document knowledge graph to enhance question answering with retrieval augmented generation. In *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4. IEEE, 2024.

[8] Shuzheng Gao, Chaozheng Wang, Cuiyun Gao, Xiaoqian Jiao, Chun Yong Chong, Shan Gao, and Michael Lyu. The prompt alchemist: Automated LLM-tailored prompt optimization for test case generation. *arXiv preprint arXiv:2501.01329*, 2025.

[9] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. ChatUniTest: A framework for LLM-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576, 2024.

[10] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[11] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 213–222, 2009.

[12] Shraddha Barke, Michael B James, and Nadia Polikarpova. Grounded Copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85–111, 2023.

[13] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[14] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. TestSpark: IntelliJ IDEA's Ultimate test generation companion. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 30–34, 2024.

[15] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, pages 419–428, 2014.

[16] Swimm. AI tools for developers: Copilot vs. Tabnine – 6 key differences, 2025. https://swimm.io/learn/ai-tools-for-developers/copilot-vs-tabnine-go-head-to-head-6-key-differences.

A APIs Utilized in Experiments

1. **Change Address v2 API**
    - **Description:** The Change Address v2 API is used to update the address for a given billing account.
    - **Endpoint:** POST https://stesbonline01:<port>/JaxRsWeb/account/{accountId}/address/v2
    - **Parameters:**
        - correlationId (HEADER, Optional): Used for tracing the call.
        - endUserId (HEADER, Must): User for mutation history.
        - accountId (PATH, Must): Billing account ID.
        - roles (QUERY, Optional)): Comma-separated list of address roles. Defaults to ADI, ADF, ADK, ADM if omitted.
        - address(JSON payload, Must): Include details like name1, name2, firstName, street, houseNo, zip, cityTown, countryCd, etc., with specific formats and lengths for each field. Among them, name1, zip, and cityTown are mandatory fields.
    - **Output:**
        - On success, the API returns HTTP Code 200, includes the newly created addressId (e.g., {"addressId": 9508150}).
        - In case of errors, default error handling applies, and an appropriate error response is returned.
2. **/ustId/v1 API**
    - **Description:** The /ustId/v1 API allows updating the VAT ID (Ust-ID) for a specific billing account.
    - **Endpoint:** POST https://stesbonline01:<port>/JaxRsWeb/account/write/{accountId}/ustId/v1
    - **Parameters:**
        - correlationId (HEADER, Optional): Used for tracing the call.
        - endUserId (HEADER, Must): User for mutation history.
        - accountId (PATH, Must): The billing account ID where the VAT ID will be updated.
        - ustId (QUERY, Must): The new VAT ID, with a maximum length of 20 characters.
    - **Output:**
        - HTTP Code 200: Indicates successful update of the VAT ID.
        - HTTP Code 404: Returned if the specified account is not found.
        - Default error handling applies in other scenarios.
3. **Read Ust-Id API**

- **Description:** The Read Ust-Id API retrieves the VAT ID (Ust-ID) for a given billing account.
- **Endpoint:** GET https://stesbonline01:<port>/JaxRsWeb/account/{accountId}/ustid/v1
- **Parameters:**
  - correlationId (HEADER, Optional): Used for tracing the call.
  - endUserId (HEADER, Must): User for mutation history.
  - accountId (PATH, Must): The billing account ID where the VAT ID will be retrieved.
  - refDate (QUERY, Optional)): Reference date in the format YYYY-MM-DD (defaults to the current date).
- **Output:**
  - HTTP Code 200: Indicates successful retrieval of the VAT ID with a JSON array containing the ustId (e.g., CHE-114.047.304).
  - HTTP Code 404: Returned if the specified account is not found.
  - Default error handling applies in other scenarios.

4. **Read Contract History API**
   - **Description:** The Read Contract History API retrieves the contract history for a specified contract.
   - **Endpoint:** GET https://stesbonline01:<port>/JaxRsWeb/contract/{contractId}/history/v1
   - **Parameters:**
     - correlationId (HEADER, Optional): Used for tracing the call.
     - endUserId (HEADER, Must): User for mutation history.
     - contractId (PATH, Must): The unique ID of the contract.
   - **Output:**
     - HTTP Code 200: Indicates successful retrieval with a JSON list containing the contract history, include information like contractId, accountId, description, timestamp, etc.
     - Default error handling applies in other scenarios.

B Example AI response for test case design

Example test cases designed by SwisscomGPT(Upload):

Prompt: Give me test cases for Read Ust-Id API with accountId 13004204
SwisscomGPT: Sure, here are the test cases for the "Read Ust-Id" API with the accountId 13004204:
Test Case ID: TC1
- **Test Case Description**: Validate successful retrieval of Ust-Id for the given accountId 13004204.
- **Test Steps**:
1. Create the request URL as '`https://s[p|i|t]esbonline01:<port>/JaxRsWeb/`

`account/13004204/ustid/v1`'
2. Execute the GET method on the above-constructed API endpoint.
3. Check the HTTP response code and message.
- **Expected Result**: HTTP status code 200 and the response body should contain the Ust-Id related to the accountId.

Test Case ID: TC2
- **Test Case Description**: Test the API's response when a non-existing accountId is supplied.
- **Test Steps**:
1. Construct the request URL as '`https://s[p|i|t]esbonline01:<port>/JaxRsWeb/account/<non-existent-accountId>/ustid/v1`'
2. Execute a GET method on the above endpoint.
3. Analyze the HTTP response code and message.
- **Expected Result**: HTTP status code 404, indicating that the accountId was not found.

Test Case ID: TC3
- **Test Case Description**: Test API response when an incorrect HTTP method (like POST) is used instead of GET.
- **Test Steps**:
1. Set the request URL as '`https://s[p|i|t]esbonline01:<port>/JaxRsWeb/account/13004204/ustid/v1`'
2. Make a POST request to the API endpoint.
3. Check the HTTP response code and message.
- **Expected Result**: HTTP status code 405 (Method Not Allowed), indicating that the POST method is not supported for this endpoint.

Test Case ID: TC4
- **Test Case Description**: Test the behavior of the API when required headers (like proper authentication tokens) are missing.
- **Test Steps**:
1. Construct the API endpoint as '`https://s[p|i|t]esbonline01:<port>/JaxRsWeb/account/13004204/ustid/v1`'
2. Make a GET request to the URL without providing required headers. 3. Analyze the HTTP response code and the response message. - **Expected Result**: HTTP status code 401 (Unauthorized), indicating that the request lacks valid authentication credentials. These test cases aim to provide comprehensive coverage of potential conditions for the 'Read Ust-Id' API.

Example test cases designed by SwisscomGPT(GPT 4-Turbo):

Here are the detailed test cases for the "Read Contract History" API using the structure and scenarios provided:
Test Case ID: TC_RCH_001 Description: Verify that the "Read Contract History" API retrieves contract history successfully for a valid contractId.
Preconditions:
    * The user "tgdgalu1" is authenticated and authorized to access the contract history.
    * A contract with contractId 1340676 exists.

Test Steps:
1. Authenticate the user "tgdgalu1" and establish a secure HTTPS connection.
2. Compose a GET request with the endpoint, replacing contractId with "1340676".
3. Include a mandatory header, endUserId, and an optional correlationId.
4. Send the GET request to the server.

Input Data:
* HTTP Method: GET
* URL: `https://sitesbonline01:<port>/JaxRsWeb/contract/1340676/history/v1`
* Headers:
  * endUserId: "tgdgalu1"
  * correlationId: "8c38500e-2de8-11ed-a261-0242ac120002"

Expected Result:
* The response is HTTP Status Code 200 OK.
* The response body contains a JSON payload with the contract's history details.

Test Case ID: TC_RCH_002 Description: Verify that the API returns an error when called with an invalid contractId.

Preconditions:
* The user "tgdgalu1" is authenticated and authorized.

Test Steps:
1. Authenticate the user "tgdgalu1".
2. Send a GET request with an invalid contractId.

Input Data:
* URL with an invalid contractId, e.g., `https://stesbonline01:<port>/JaxRsWeb/contract/invalid_contract/history/v1`.

Expected Result:
* The response is a HTTP Status Code 400 Bad Request or 404 Not Found, indicating the contractId is not valid.

Test Case ID: TC_RCH_003 Description: Verify that the API returns an error when the mandatory endUserId header is missing.

Preconditions:
* The contract with contractId 1340676 exists.

Test Steps:
1. Establish a secure HTTPS connection.
2. Prepare and send a GET request lacking the endUserId header.

Input Data:
* Headers without endUserId.

Expected Result:
* The response is HTTP Status Code 400 Bad Request due to the missing required header.

Test Case ID: TC_RCH_004 Description: Verify that the API handles unauthorized access, missing user credentials (Security Testing).

Preconditions:
* Invocation of the API with missing or invalid user credentials.

Test Steps:
1. Attempt to connect to the server without user credentials.
2. Send the GET request with required headers except for the authorization token.

Input Data:
    * Correct header information but no authorization token.
Expected Result:
    * The response is HTTP Status Code 401 Unauthorized or 403 Forbidden.
Test Case ID: TC_RCH_005 Description: Verify the API behavior under heavy load conditions (Performance Testing).
Preconditions:
    * The contract with contractId 1340676 exists, and the user "tgdgalu1" is prepared for load testing.
Test Steps:
    1. Use a load testing tool to simulate multiple requests to the API.
    2. Monitor the response time and error rates of the API.
Input Data:
    * Utilize a load testing tool to generate a large volume of valid requests.
Expected Result:
    * The API maintains a consistent response time and has an error rate within acceptable limits under heavy load.

Ensure to replace <port> with the actual port number before running the test cases. Moreover, during performance testing, ensure that the test environment mirrors the production environment as closely as possible and that the test does not impact actual users or production systems.

Example test cases designed by Cursor Chat(Claude 3.5):

I'll create detailed test cases with specific input data and steps for the "/ustId/v1" API.

Positive Test Cases

TC1: Basic Valid Request

Description: Verify successful retrieval of VAT ID with valid inputs

Test Data:

- AccountId: 13004204
- Headers:

```
endUserId: "tgdmith4"
correlationId: "8c38500e-2de8-11ed-a261-0242ac120002"
```

Test Steps:

1. Set up request with valid headers
2. Send GET request to endpoint
3. Capture response code and body
4. Validate JSON structure
5. Verify ustId format

Expected Results:

- Status code: 200
- Response contains valid ustId
- JSON format is correct
- Response time < 1 second

TC2: Request with Optional Parameters

Description: Verify request works with all optional parameters

Test Data:

- Same as TC1 plus:

```
refDate: "2024-12-31"
```

Test Steps:

1. Set up request with all parameters
2. Execute GET reque
3. Validate response
4. Compare ustId with expected value for given date

Expected Results:

- Status code: 200
- Correct historical ustId returned
- All response fields present

Negative Test Cases

TC3: Missing Mandatory Header

Description: Verify error handling for missing endUserId

Test Data:

- AccountId: 13004204
- Headers:

```
correlationId: "8c38500e-2de8-11ed-a261-0242ac120002"
// endUserId intentionally omitted
```

Test Steps:

1. Set up request without endUserId
2. Send GET request
3. Verify error response

Expected Results:

- Status code: 400
- Error message: "Missing mandatory header: endUserId"

TC4: Invalid Account ID

Description: Test handling of non-existent account

Test Data:

- AccountId: 99999999
- Headers: Same as TC1

Test Steps:

1. Send request with invalid AccountId
2. Capture response
3. Verify error details

Expected Results:

- Status code: 404
- Error message indicating account not found

Security Test Cases

TC5: Unauthorized Access

Description: Verify authentication requirement

Test Data:

- AccountId: 13004204

- No authentication token
- Headers: Same as TC1

Test Steps:

1. Remove authentication token
2. Send request
3. Verify security response

Expected Results:

- Status code: 401
- Authentication error message

TC6: Cross-Account Access

Description: Verify account access restrictions

Test Data:

- AccountId: Different from authorized user
- Valid authentication token
- Headers: Same as TC1

Test Steps:

1. Use token for Account A
2. Try to access Account B's ustId
3. Verify access denied

Expected Results:

- Status code: 403
- Access denied message
- Data Validation Cases

TC7: Invalid Date Format

Description: Test date format validation

Test Data:

- AccountId: 13004204
- Headers: Same as TC1
- Query params:

```
refDate: "31-12-2024" // incorrect format
```

Test Steps:

1. Send request with invalid date
2. Verify error handling

Expected Results:

- Status code: 400
- Date format error message

TC8: Future Date

Description: Test future date handling

Test Data:

- AccountId: 13004204
- Headers: Same as TC1
- Query params:

```
refDate: "2030-12-31"
```

Test Steps:

1. Send request with future date
2. Verify response

Expected Results:

- Status code: 400
- Future date error message

Performance Test Cases

TC9: Response Time

Description: Verify API performance

Test Data:

- Same as TC1
- Number of requests: 100

Test Steps:

1. Send 100 sequential requests
2. Measure response times
3. Calculate average, min, max times
4. Check for errors

Expected Results:

- All responses $<$ 1 second
- No errors
- Consistent response times

TC10: Concurrent Access

Description: Test concurrent request handling

Test Data:

- Same as TC1
- 50 concurrent requests

Test Steps:

1. Prepare 50 concurrent requests
2. Execute simultaneously
3. Monitor responses
4. Check for failures

Expected Results:

- All requests successful
- No timeout errors
- Response time $<$ 2 seconds

Integration Test Cases

TC11: Data Consistency

Description: Verify ustId consistency across systems

Test Data:

- AccountId with known ustId
- Headers: Same as TC1

Test Steps:

1. Get ustId from API
2. Query backend database
3. Compare values
4. Verify timestamps

Expected Results:

- Matching ustId values
- Consistent timestamps
- No data discrepancies

Each test case includes:

- Clear description
- Specific test data
- Detailed steps
- Expected results

Additional considerations:

- Log all test executions
- Capture response times
- Document any deviations
- Track test coverage

These test cases should be executed in different environments (DEV, TEST, PROD) with appropriate data sets for each environment.

C Test cases designed by AI

API	**Change address v2**	**/ustId/v1**	**Read Ust-Id**	**Read Contract History**
TC1	Ensure successful execution	Verify successful retrieval	Test successful retrieval	Test successful reading
TC2	Test with a non-existent accountId	Check with a non-existing accountId	Test with a non-existent accountId	Test with an invalid contractId
TC3	Using GET instead of POST	Using POST instead of GET	Using POST instead of GET	Test with missing contractId
TC4	Test with insufficient required fields	Give incorrect correlationId	Test with incorrect authorization header	Using POST instead of GET

Table 14: Test cases designed by Test Case Designer(Upload) instance

Case Type	Change address v2	/ustId/v1	Read Ust-Id	Read Contract History
Valid TC	**TC1**: Successful address change **TC2**: Missing authentication token **TC3**: Test with non-existing accountID **TC4**: Missing mandatory fields in JSON **TC5**: Invalid data types in JSON	**TC1**: Verify successful write **TC2**: Test with non-existing accountId **TC3**: Test without an endUserId header **TC4**: Test with invalid ustId format	**TC1**: Verify successful retrieval **TC2**: Test with an invalid accountId **TC3**: Test without an endUserId header **TC4**: Test without authentication credentials	**TC1**: Verify successful retrieval **TC2**: Test with an invalid contractId **TC3**: Test without an endUserId header **TC4**: Test without authentication credentials
Non-API TC	**TC6**: Verify system behavior for XSS attack **TC7**: Verify API behavior under high load	**TC5**: Verify API behavior under high load	**TC5**: Verify API behavior under high load	**TC5**: Verify API behavior under heavy load conditions

Table 15: Test cases designed by SwisscomGPT(GPT 4-Turbo)

API	Valid TC	Redundant TC	Non-API TC	Wrong TC
Change address v2	**TC1**: Basic Address Change **TC2**: Complete Address Update **TC3**: Missing Mandatory Fields in JSON **TC4**: Invalid Account ID **TC5**: Field Length Validation **TC6**: Special Characters **TC7**: Authorization Test **TC12**: History Tracking	**TC8**: Cross-Account Access	**TC9**: Load Testing **TC10**: Sequential Updates	**TC11**: Data Consistency

API	Valid TC	Redundant TC	Non-API TC	Wrong TC
/ustId /v1	**TC1**: Basic UstId Update **TC2**: Update with Optional Fields **TC3**: Invalid Account ID **TC4**: Missing Mandatory Header **TC5**: UstId Format Validation **TC6**: Maximum Length Tests **TC7**: Authorization Verification	**TC8**: Cross-Account Access **TC13**: Special Characters	**TC9**: Load Testing **TC10**: Rapid Sequential Updates	**TC11**: System Synchronization **TC12**: Audit Trail
Read Ust-Id	**TC1**: Basic Ust-Id Retrieval **TC3**: Invalid Account ID **TC4**: Missing Mandatory Header **TC5**: Unauthorized Access **TC6**: Insufficient Permissions **TC13**: Account Status Changes	**TC2**: Request with Different Account **TC7**: Ust-Id Format Validation **TC8**: Special Characters	**TC9**: Response Time **TC10**: Concurrent Access	**TC11**: Data Consistency **TC12**: Audit Trail
Read Contract History	**TC1**: Basic Contract History Retrieval **TC3**: Invalid Contract ID **TC4**: Missing Mandatory Header **TC5**: Unauthorized Access **TC6**: Insufficient Permissions **TC7**: Contract Data Completeness **TC8**: Large History Dataset	**TC2**: Contract with Multiple History Records **TC13**: Special Characters	**TC9**: Response Time Verification **TC10**: Concurrent Access	**TC11**: Cross-System Validation **TC12**: Audit Trail Verification
Total	28	8	8	7

Table 16: Test cases designed by Cursor Chat(Claude 3.5)