UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Karl-Andreas Meus

Facilitating the Automation of Use Case Specifications and Test Case Generation by Developing an LLM-Powered Tool

Bachelor's Thesis (9 ECTS)

Supervisor(s): Marinos Georgiadis, PhD

Tartu 2025

Facilitating the Automation of Use Case Specifications and Test Case Generation by Developing an LLM-Powered Tool

Abstract:

Software projects are often delayed or experience increased costs due to unclear or incomplete requirements. This thesis addresses these challenges by introducing ReqFlowly, a web-based application that automates key aspects of the requirements engineering process. ReqFlowly utilizes a Large Language Model (LLM), selected through a comparative evaluation. The model is guided by a formal use case grammar and structured prompts to help users transform initial stakeholder requirements into structured artifacts: domain entities, precise use case specifications, and comprehensive test cases. The proposed solution aims to minimize manual effort during the early phases of software development. It facilitates the production of software specifications that are more complete, consistent, and precise, thereby enhancing the overall efficiency and reliability of the engineering process.

Keywords: Requirements engineering, artificial intelligence, LLM, use case, test case

CERCS: P170 - Computer science, numerical analysis, systems, control

LLM-põhise tööriista arendamine kasutusjuhtumite spetsifikatsioonide ja testjuhtumite genereerimise automatiseerimiseks

Lühikokkuvõte:

Ebaselged ja mittetäielikud nõuded põhjustavad tarkvaraprojektide valmimisel tihtipeale viivitusi ja lisakulusid. Bakalaureusetöö eesmärk oli luua veebirakendus ReqFlowly, mis aitab eelnimetatud probleeme lahendada automatiseerides tarkvaranõuete analüüsi protsessi. Rakendus kasutab tehisintellekti ning reeglipõhiseid heuristikaid, et aidata kasutajatel tuvastada nõuetest domeeniobjektid ning genereerida kasutus- ja testjuhtumid. Lahendus vähendab oluliselt manuaalset tööd, parandab nõuete täpsust ning muudab kogu analüüsi protsessi usalduvsäärsemaks ja kuluefektiivsemaks.

Võtmesõnad: Nõuete analüüs, tehisintellekt, LLM, kasutusjuhtum, testjuhtum

CERCS: P170 - Arvutiteadus, arvanalüüs, süsteemid, kontroll

Table of Contents

1. Introduction

In the process of software development, one of the most important steps is to ensure a clear understanding of system functionalities before any significant development starts [1]. Bajceta et al. [2] emphasize that as software systems become increasingly complicated, traditional manual approaches to elicit and validate requirements become inefficient and more prone to oversight. While requirements engineering methods have also been evolving with time, there remains a need for a more advanced, Artificial Intelligence (AI) assisted solution that can transform initial stakeholders needs into precise, consistent and unambiguous system specifications.

Despite the rapid progress of AI across many domains of software development, requirements elicitation and specification continue to be mostly done by hand. This manual process is prone to human error, can take considerable time, and may cost companies a lot of money if misunderstandings, often due to ambiguous requirements, are discovered too late in the development cycle [1]. According to the Standish Group's Chaos Report, inadequate requirements are among the primary contributions to project delays and budget overruns, leading to billions of dollars in losses worldwide [3]. The identified shortcomings provide a compelling case for developing a tool that helps to uncover such ambiguities and inconsistencies early and to ensure a more stable foundation for the entire development process.

This thesis addresses these challenges by creating a web-based application that uses AI to assist users on every step of the requirements analysis workflow, from identifying key domain entities to creating and managing use cases and test cases. By leveraging AI through a sequence of guided prompts, the tool systematically processes user-provided information to generate domain entities, then employs a defined use case grammar to produce unambiguous use case specifications, and finally derives corresponding test cases. This approach speeds up key aspects of the requirements engineering process and significantly enhances traceability as well as clarity, consistency, and unambiguity of the generated specifications throughout the software lifecycle.

The primary aim of this thesis is to design and develop an innovative AI-powered application that will assist users in the critical process of transforming initial stakeholder requirements (such as user stories) into precise, unambiguous use case specifications and into derived test cases.

More specifically, the main objectives of this thesis are:

1. To automate the generation of detailed use case specifications that are precise, consistent, unambiguous, and conform to established modeling standards. This is achieved through an AI-powered functionality that guides the user to construct these specifications using a highly detailed and thorough prompt, structured according to a published use case grammar. This approach also uses key domain entities and their attributes, which are established through an AI-driven analysis of initial user requirements, such as user stories.
2. To automate the generation of comprehensive test cases, designed to cover both positive scenarios (verifying expected behavior with valid inputs and conditions) and negative scenarios (evaluating system robustness and error handling with invalid inputs or unexpected conditions). These test cases will be logically derived from the generated use case specifications to ensure thorough validation of the specified functionalities.
3. To design and implement a straightforward and intuitive web-based interface that lets users to effectively manage the requirements engineering workflow and its artifacts. This interface will support: a. Adding and organizing initial stakeholder requirements. b. Reviewing and refining the AI-derived domain entities. c. Initiating, overseeing, and customizing the generation process for use cases and test cases. d. Managing and interacting with all the generated artifacts (domain objects, use cases, test cases). e. Clearly visualizing and navigating the traceability links that are automatically established and maintained between initial requirements, domain objects, use case specifications, and test cases, ensuring lifecycle consistency.
4. To conduct an initial evaluation of the developed web application. This evaluation will focus on assessing its effectiveness in automating key requirements specification tasks, as well as the quality of the generated artifacts (use cases and test cases).

2. Theoretical overview and related work

This chapter provides a theoretical overview of the core concepts of the requirements engineering process. This overview includes requirements elicitation, analysis, specification and validation. The chapter mainly focuses on techniques such as domain modeling (to identify and define main domain entities and their attributes), use case modeling and specification (to detail system behavior through use cases), and test case design principles (leading to the development of test cases for system verification). The detailed overview of these particular topics is important because of their role in the application developed in this thesis. As outlined in the objectives, the application's main goal is to automate the creation of domain entities, use cases and test cases from initial requirements.

Additionally, the chapter investigates the possibilities of using AI in the process of automating the creation and management of these artifacts. Finally, the chapter analyses related work. This involves a comparison of existing solutions that have integrated AI into the requirements engineering workflow, particularly for automating the generation of use cases and test cases. This comparison shows the capabilities and limitations of current available tools in relation to the proposed solution.

2.1 Theoretical foundations

The automation of requirements analysis and artifact creation proposed in this thesis relies on well-established theoretical practices of requirements engineering. These include domain modeling, use case specification and test case development. This section provides an overview of these concepts and explains their role in improving the software development process by making requirements clearer.

2.1.1 Requirements engineering fundamentals

Requirements engineering is a structured process of eliciting, analyzing, documenting, validating and managing software requirements. It is a critical phase in software engineering, given that the

success and the quality of software systems are heavily dependent on the accuracy, completeness, and clarity of the defined requirements [4].

Software development involves various types of requirements that engineers must identify and manage. Among these, functional requirements are central, as they specify what the system is expected to do. These requirements form a direct basis for use case generation because they represent in detail how the system should behave. On the contrary, non-functional requirements focus on qualities such as performance, usability, reliability, and security and are typically evaluated throught different techniques [5]. Therefore, the precise identification and formulation of functional requirements is foundational for the automated analysis and specification activities, including the artifacts targeted in this thesis.

Despite its importance, the requirements engineering process is not withouts its challenges. Common issues such as ambiguities in natural language descriptions, inconsistent documentation, incomplete information, and poor requirements traceability can lead to misunderstandings and missed requirements. These problems frequently cause delays, budget overruns, and failures to meet stakeholder expectations [4]. To address these challenges, it is important to use structured and reliable techniques that improve precision, consistency and verifiability of requirements throughout the software lifecycle.

2.1.2 Domain modeling in requirements engineering

Domain modeling is an important phase in requirements engineering aimed at understanding and representing the key concepts, such as domain entities, within the system's problem space [6]. These entities are the fundamental building blocks that the system will manage, interact with, or track information about [7]. Manually identifying and defining a complete set of domain entities from large textual requirements can be a big challenge. Issues such as subjective interpretation, oversight of important concepts and maintaining consistency across a large set of requirements may all potentially impact the quality of subsequent design and development phases. Identifying domain entities typically involves analyzing functional requirements and other system documentation, often by looking for significant nouns that represent persistent concepts relevant

to the domain, such as “customer”, “product,”, “wishlist” or “order” in various systems. It is also important to extract the attributes of the domain objects, such as “name”, “description” or other properties that define or describe the entity’s state and behaviour. The correct identification and definition of these entities and their attributes are critical for the system, because they serve as a direct basis for subsequent modeling activities, such as use case specification and database design. Figure 1 shows an example of three domain entities and their attributes identified in the context of an online store platform.

Domain object	Attributes
Customer	id, name, email, isActive, lastActiveAt, role, createdAt
Product	id, name, description, price, availability, listedAt, updatedAt
Wishlist	id, customerId, createdAt, updatedAt

Figure 1. Domain entitites and their attributes in an online store context.

The figure includes domain entities “customer”, “product” and “wishlist”. Each entity is described by a set of attributes that provide insights into its characteristics and potential behavior within the system which can be used for use case generation. By extracting and polishing these domain entities, the requirements engineers gain a clearer understanding of what the software must support.

2.1.3 Use case modeling and specification

Use cases are a commonly used technique in requirements engineering for capturing and specifying system functionality. They describe how actors, which can be users or other systems, interact with the system under development to achieve specific goals. Originally introduced by Jacobson et al. [8] in the context of object-oriented analysis and design, use cases define sequences of actions performed by an actor and the system's responses, thereby clarifying functional requirements by illustrating various interaction paths, including the main success scenario and any alternative or exception paths.

Presenting requirements as use cases helps stakeholders visualize system behaviour in realistic scenarios. Each use case typically represents a single goal for the actor, ensuring that

specifications remain organized and easy to interpret [8]. Additionally, use cases help to identify missing or ambiguous requirements early, reducing the potential miscommunication between developers, testers, and users [8]. Figure 2 shows an example use case for the functionality "Add item to wishlist". This example illustrates a typical sequence of steps, including actors, triggers, preconditions, exceptions and postconditions.

Element	Detail
Name	Add Item to Wishlist
ID	UC01
Description	A user finds an item in the eStore that they want to add to a list so that they can return to the item at a later date.
Actor	Registered eStore Customer
Triggers	Customer clicks "Add to Wishlist" while viewing a product.
Preconditions	1. Product exists and is displayed. 2. Customer is authenticated (see UC-ACC-01 Log In).
Postconditions	1. Item is linked to the customer's wishlist and stored persistently. 2. Timestamp of addition is recorded for sorting.
Main Flow	1. Customer selects "Add to Wishlist" on the product page. 2. System verifies that the product is not already on the customer's wishlist. 3. System creates a wishlist-item record (product-ID, customer-ID, date-added). 4. System confirms: "Item added to your wishlist."
Alternate Flow	AF01 - Customer not logged in 1. System presents Log In / Sign Up options. 2. On successful log-in (UC-ACC-01), system returns to Main Step 2. AF02 - Item already on wishlist 1. System notifies: "This item is already on your wishlist." 2. Use case ends.
Exception Flow	EX 1 - Database or wishlist service unavailable 1. System shows error: "Could not add item right now—please try again later." 2. Use case ends with failure state recorded in logs.

Figure 2. Use case for the functionality "Add item to wishlist" [9].

Use cases provide a strong foundation for development and testing. For example, each step in a use case can directly inform the creation of test scenarios and test steps. This helps to ensure that test cases closely reflect the intended system behavior and user goals [10]. The structured nature of well-defined use cases, particularly when guided by principles such as those found in use case

grammars (discussed in the next section), can further increase their usefulness for automated processing, including the generation of test cases.

2.1.4 The role of use case grammars for precision

While detailed use case specifications add significant structure to functional requirements, the natural language used in their descriptions can still lead to ambiguities, inconsistencies or missing information. To address these issues and improve the clarity and precision of use cases, a use case grammar, structured use case patterns or a structured use case template can be utilized [11]. A use case grammar consists of a set of rules, constraints, and predefined structure for expressing the different elements of a use case, particularly the steps within its flow of events.

The implementation of this grammar provides several advantages. Firstly, it promotes greater precision and significantly reduces ambiguity by restricting how actions and interactions are described, resulting in clearer and more understandable specifications. Secondly, it enforces consistency across all use cases developed for a system, ensuring that similar concepts are expressed in an uniform manner. This consistency and structured format also make grammar-constrained use cases more suitable for automated processing and analysis. Software tools can more readily parse, interpret, and utilize these use cases for tasks, such as automated validation, model generation and, most relevant to this research, the automated generation of corresponding test cases.

The approach used in this thesis for generating structured and unambiguous use case specifications directly leverages a use case grammar published by Georgiades [12]. This established grammar provides the foundation for the AI-powered mechanism, introduced later in this thesis, to generate high-quality use case artifacts.

2.1.5 Test case design

Software testing is an essential phase in the software development lifecycle. Its main goal is to evaluate and verify whether a software product or application behaves as intended [13]. Effective software testing helps to identify any software issues early in the development process, reducing

the cost and effort required for fixes and helping ensure that the final product meets the specified requirements and user expectations. To support this process, test cases are commonly used.

Use cases describe how users interact with the system at a higher level but test cases are concrete executable steps to verify that the system actually behaves as expected. Use cases focus on what the system should do from the user's perspective, whereas test cases validate whether the system actually does it. A test case typically contains preconditions, inputs, actions, and expected outcomes [14]. These components help the tester systematically evaluate the system's functionality and confirm that all requirements are fulfilled. Clear and well-structured test cases help uncover defects early, enabling the development team to address issues before they escalate into more significant problems [14].

However, the manual creation of a sufficiently comprehensive and effective suite of test cases is often a demanding process, presenting several practical challenges. Designing tests to ensure thorough coverage of all functional paths, including various positive and negative scenarios, can be very time-consuming and a difficult challenge. This effort is also susceptible to oversight, potentially leading to gaps in testing or inconsistencies in the test documentation. Furthermore, maintaining consistency between a large set of test cases and frequently evolving requirements can become a significant maintenance overhead [14].

When test cases are derived directly from use cases, they maintain a clear link to the requirements set out by stakeholders. Each use case scenario can naturally be transformed into one or more test cases, preserving traceability from requirements through to implementation and testing. This connection is especially useful in agile or iterative environments, where requirements often change. In such cases, updating the use case model can automatically guide updates to the related test cases, making the testing process more responsive and efficient [14]. Figure 3 provides an example of a test case based on the “Add item to wishlist” use case shown earlier. It has the required preconditions, the sequence of user actions, the expected system behaviour, and the postconditions that verify the intended outcome.

Element	Details
ID	TC-UC01-01
Name	Add Item to Wishlist (Happy Path)
Related use case	UC01 – Add Item to Wishlist
Test objective	Verify that a logged-in customer can add a product to an empty wishlist and receives confirmation.
Preconditions	1. Customer account CUST-123 exists and is logged in. 2. Product SKU-987 is visible on the product page. 3. SKU-987 is not already on CUST-123's wishlist. 4. Wishlist service and database are online.
Test Data	Customer ID: CUST-123 Product ID: SKU-987
Steps	1. Customer clicks "Add to Wishlist" on the SKU-987 product page. 2. System checks whether SKU-987 is already on CUST-123's wishlist. 3. System inserts a new wishlist record (CUST-123, SKU-987, timestamp). 4. System displays toast/alert: "Item added to your wishlist."
Expected results	• Database table WishlistItems contains one new row with CustomerID = CUST-123, ProductID = SKU-987, and a non-null DateAdded. • UI confirmation message matches exact text (or approved copy spec). • No error or warning appears in browser console or server logs.
Postconditions	SKU-987 is now stored and retrievable from CUST-123's wishlist; timestamp can be used for ordering.

Figure 3. Test case "Add item to wishlist" derived from UC01.

2.1.6 The role of AI in generating use cases and test cases

Recent advancements in natural language processing and machine learning have made it possible to automate parts of the more repetitive parts of the requirements engineering process, particularly the generation of use cases and test cases [15]. By analyzing requirements written in plain language, AI tools can identify key elements such as actors, system actions, and data flows, and generate structured use cases that describe how users are expected to interact with the system [16].

Once use cases are defined, the same tools can help to create corresponding test cases by extracting input conditions, expected outcomes, and potential edge cases from the same requirement set. The test cases are created directly from the same requirements, which makes it easier to track which parts of the system are being tested and why [17].

In addition, AI can also adapt to requirements changes. As changes are made to requirements, the AI can suggest updates to existing use cases and test cases, ensuring that the documentation remains consistent. This reduces the need for manual corrections and helps teams keep documentation up to date and to always work with updated and consistent use cases and test cases [18].

By integrating AI into the requirements and testing process, teams can speed up documentation, reduce human error, and improve overall software quality. This kind of automation addresses many of the long-standing pain points in manual requirements engineering and test creation and contributes to more stable and maintainable software systems [18].

2.2 Related Work

Before turning to the proposed implementation, it is useful to examine the software that already attempts to tackle the problems defined in the introduction chapter. This chapter provides an overview of the three closest solutions to the issues. There is no existing software that solves all defined difficulties but three web-based tools, Copilot4DevOps [19], ScopeMaster [20], and Spec2Test [21], come to the closest. They are all similar to the proposed solution in terms of the step by step AI-supported requirements workflow functionality. However, none of them fully delivers the fully integrated, customizable pipeline from raw requirements to traceable test cases. Most importantly, only Copilot4DevOps was found, after a thorough review of existing tools, to allow the automatic converstion of textual requirements into use case specifications. The following review focuses on their capabilities, limitations and how they differ from the approach presented in this work.

2.2.1 Copilot4DevOps

Copilot4DevOps [19] is an AI-powered assistant integrated with Azure DevOps that automates several requirements-related tasks, for example generating use cases and test cases. It also offers features like requirement quality analysis using heuristics, prompting users if the tool finds any

ambiguities. The tool is particularly beneficial for users already using the Azure DevOps ecosystem, as it can be integrated into existing workflows without requiring additional setup.

However, there are several limitations with the Copilot4DevOps tool. The requirements used for automation cannot be unstructured or incomplete, because they may lead to suboptimal artifacts. Additionally, to use the tool, the users need to have an Azure DevOps license plus a Copilot4DevOps license. If the user also wants to customize the prompts used for generating the artifacts, they will need to upgrade the Copilot4DevOps license. All of this may be too expensive for smaller teams or for those not already using the Azure ecosystem.

In contrast, the proposed solution addresses these gaps by allowing users to work with unstructured or incomplete requirements. The system detects domain entities, as the very first step, before generating any use cases. And then a specific set of use cases are developed for each domain entity following a dedicated use case grammar [12] and the related prompts. This ensures a structured approach that covers all the different types of use cases for each domain entity and that all the parts (especially the main and alternate flows) of a use case are fully covered by the relevant rules. Additionally, all the prompts can be enhanced by adding more context to them and the tool is more cost-effective, making it accessible to a wider audience, including to teams outside the Azure DevOps environment.

2.2.2 ScopeMaster

ScopeMaster [20] is an AI-assisted requirements analysis tool that uses rule-based natural language processing to assist with early-stage requirement validation. Its main features are ambiguity detection, function point estimation, and the generation of basic test cases. While ScopeMaster is useful for identifying poorly defined requirements, it lacks support for AI-driven interpretation, interactive use case generation, and real-time feedback or customization. In addition, ScopeMaster's pricing can be a limiting factor for smaller teams or projects, especially considering the additional tools that might still be needed to achieve the full coverage of the requirements analysis workflow.

In comparison, the proposed tool addresses these limitations by offering a more interactive, intelligent and cost-effective solution. Unlike ScopeMaster, which relies on static rule-based NLP, the proposed tool uses AI-powered interpretation to dynamically extract domain entities and handle unstructured input, generate structured use cases and produce both positive and negative test cases. Users have full control over customizing the workflow. In addition, the developed solution keeps the project costs low by using the Gemini API efficiently, making it far more accessible for teams of any size. It is important to highlight that the proposed application utilizes customized prompts at every step of the process, which have been specifically developed and iteratively refined to achieve optimal results. Another significant aspect, as already mentioned, is that the proposed tool leverages a published use case grammar [12] as input in the second step along with its associated prompt, thereby enhancing the rigor, specificity, and reliability of the use case specifications generated by the LLM. These combined features distinctly differentiate the approach and tool from others currently available.

2.2.3 Spec2Test AI

Spec2Test AI [21] presents itself as a detect-prevention engine platform. The workflow used on the platform starts with the input of user stories, meaning the workflow assumes that the user stories are already in place. The user stories are then analyzed by LLM models that score each story against 32 best practice quality metrics, including metrics like INVEST, SMART and REAL. The tool then highlights lexical, semantic and pragmatic ambiguities and proposes rewritten wording. After the user stories have been finalized by the user, the tool generates Behaviour-Driven Development (BDD) feature files, language neutral pseudocode for tests and a suite of functional, negative-path, edge-case and OWASP-aligned test cases. All generated artifacts are linked back to their source user stories for traceability.

Spec2Test AI workflow begins only after well-formed user stories exist, it cannot operate with vague and not finalized requirements, therefore limiting its usefulness during the explanatory phases of a project. In contrast, the tool developed in this thesis can parse unstructured stakeholder statements and construct structured use cases before moving on to test case generation, allowing it to support a much broader slice of the requirements-elicitation workflow. Because the use case model makes every alternative and exception path explicit, the resulting

tests probe a wider spectrum of system behaviour than tests derived directly from user stories. In addition, the proposed implementation gives room for customization. Spec2Test AI relies on hard-coded AI language models which cannot be prompted to act differently based on the user's needs. The proposed solution fills these gaps by supporting unstructured input from the outset and allows to add additional context and instructions to the AI, which could result in better results if the user has specific needs. Affordability is another differentiator, Spec2Test AI is marketed on a subscription model. Its lowest tier is fairly expensive to use, priced at 50 USD per month per seat plus onboarding and consumptiong charges, placing it in the enterprise market bracket.

While Copilot4DevOps, ScopeMaster and Spec2Test offer partial solutions to AI-assisted requirements engineering, they fall short of delivering a fully integrated, customizable and accessible pipeline. The tool developed in this thesis distinguishes itself by addressing unstructured inputs from the earliest stages, enforcing use case grammar for higher accuracy and completeness, and supporting detailed customization through customizable/tunable AI prompts and all of it while maintaining affordability and usability across different project scales.

3. Requirements of the application

This section outlines the functional and non-functional requirements of the developed web application. Functional requirements are organized according to the different features within the application. Non-functional requirements on the other hand define the quality attributes and technical constraints the application must satisfy. Both requirements serve as a foundation for system architecture, implementation and evaluation.

3.1 Functional requirements

The functional requirements are organized based on the different features of the application to provide clear navigation and functionality. Each requirement is labeled with "FR" prefix to distinguish it from non-functional requirements and to support consistent referencing throughout the development and evaluation process.

Landing and navigation

FR1. The landing page must provide/display the purposes and key features of the application.

FR2. The landing page must provide navigation to the login page.

User management and authentication

FR3. Users must be able to log in using email or with a Google account.

FR4. New users must have the option to register an account.

FR5. Users must be able to reset their password.

Project management

FR6. Users must be able to create new projects with a name and a description.

FR7. Users must have an overview of all created projects.

FR8. Users must be able to select and open existing projects.

FR9. Users must be able to filter projects by name, creation date and last updated date.

FR10. Users must be able to view, edit and delete projects.

FR11. Users must be able to upload requirements as a text to a project.

FR12. Users must be able to upload requirements as a PDF file to a project.

FR13. Users must be able to open requirements for a detailed view.

Requirements management

FR14. Users must be able to view, edit and delete uploaded requirements.

FR15. Editing requirements, must trigger a warning notification indicating that other artifacts derived from the requirement may need to be recreated due to the changes.

FR16. Users must be able to navigate to domain objects page from the opened requirements.

Domain objects extraction

FR17. Users must be able to view and select requirements for domain object creation out of all the uploaded requirements.

FR18. Requirements must be automatically prefilled when the user navigates to the domain objects page from a selected requirement.

FR19. Users must be able to manually create new requirements with a text insert.

FR20. Users must be able to manually create new requirements with a PDF upload.

FR21. Users must be able to add context to the prompt for AI processing.

FR22. Users must be able to view, edit, delete and save the generated domain objects and their attributes.

FR23. Users must be able to add new domain objects and attributes to the generated list.

FR24. Saving domain objects must redirect the user to the use cases and test cases page.

Use case and test case generation

FR25. Users must have an overview of all created batches of domain objects and attributes.

FR26. Users must be able to open a batch for a detailed view.

FR27. Users must be able to generate use cases for each domain object and its attributes.

FR28. Users must be able to generate test cases derived from the created use cases.

FR29. Users must be able to view, edit and delete existing use cases.

FR30. Users must be able to view, edit and delete existing test cases.

FR31. Users must be able to navigate directly to the export page.

Export and documentation

FR32. Users must be able to select different artifacts for export.

FR33. Users must be able to export the selected items as a PDF file.

3.2 Non-functional requirements

The non-functional requirements specify the expected qualities and constraints under which the application operates. Each requirement is prefixed with "NFR" to clearly distinguish it from functional requirements and to support traceability throughout the development process. According to the use case grammar, the proposed solution must be able to generate 18 use cases and the chapter 2 (introduction, label 2) shows that a well-formed use case body averages to about 250 words. Allowing double that size provides a buffer for unusually complex scenarios without inflating LLM costs. These constraints are specified in the non-functional requirements.

NFR1. The application must be globally accessible via the Internet.

NFR2. Automatic updates of the application must occur following code deployment to GitHub.

NFR3. All code must be structured in a way that makes it testable.

NFR4. Essential HTML elements must include ARIA labels to ensure compatibility with screen readers.

NFR5. The generation of content with AI must not take more than 1 minute.

NFR6. The application must be fully compatible with major web browsers like Google Chrome, Mozilla Firefox, Microsoft Edge, and Safari.

NFR7. The application's design and user interface must appear professional.

NFR8. Navigation and interactions within the application must be intuitive and user-friendly.

NFR9. The application must use open-source software with a substantial community to ensure long-term support and maintenance.

NFR10. Initial load times, page transitions and data filtering operations must complete in under 3 seconds.

NFR11. User authentication and authorization must follow standard security protocols.

NFR12. The application should have sufficient logging for debugging and monitoring purposes.

NFR13. The LLM service must operate on a pay-per-request basis, with zero pre-payment or fixed monthly charge.
NFR14. The LLM maxium output token limit must be sufficient to generate eighteen use cases in one output.
NFR15. The LLM must support generating individual use cases with an average length of approximately 500 words.

All functional and non-functional requirements defined in this chapter were successfully implemented as part of the final application, ensuring completeness and alignment with the original specifications. These results show that the project has delivered the complete, integrated workflow envisioned in the beginning of the thesis.

4. Architecture

The system architecture for this project was designed with priorities on scalability, reliability and developer productivity. The selection of technologies also took NFR9 into account. To meet these needs, the system follows the three-tier architecture, a widely used and well-established software application architecture that organizes applications into three logical and physical computing tiers: the presentation tier, the application tier and the data tier [22].

The three-tier architecture offers several benefits, such as improved maintainability, better scalability and stronger system security. Firstly, it enables the separation of concerns, making each layer independently manageable and testable. In the future, this allows the application tiers to be further developed and maintained independently, allowing multiple developers to work on each tier without affecting each other. Secondly, each tier can be scaled up separately when needed. For example if the application's logic becomes a performance bottleneck, the application tier alone can be scaled up without affecting other tiers of the system. Finally, the architecture improves system security by isolating the data access layer, making the application tier function as an internal firewall, preventing SQL injections and other malicious exploits.

4.1 Presentation tier

The presentation tier, responsible for the user interface, was developed using React [23]. React is a very popular and powerful open-source JavaScript library. It is specifically created for building dynamic, responsive, and user-friendly applications. It was the choice for the presentation tier because React allows developers to manage complex UI states very efficiently and create highly reusable components. These criteria were needed for building the interactive, step-by-step workflow in the project.

In addition, according to the Stack Overflow Developer Survey 2024 [24], React is the most widely used JavaScript framework by a significant margin. Its large community ecosystem provides up-to-date libraries and tooling used for the project and ensures that the potential development challenges can be resolved efficiently with abundant resources and support always

available. This boosts the developer productivity and simplifies testing, further supporting its selection as the optimal solution.

Alternatives AngularJS [25] and Vue.js [26] were also considered, but AngularJS, while being with a good reputation, is more suited for larger scale, enterprise level applications. For the project developed in this thesis, simply this kind of added complexity that AngularJS provides was not needed. Vue.js, while being more simple than AngularJS, has a smaller community compared to React, resulting in fewer third-party libraries, resources, and long term support. Given the need for a highly maintainable and extensible solution and considering also the author's experience with all of the three technologies, React.js was determined to be the most appropriate choice.

4.2 Application tier

The application tier, responsible for handling the business logic, was developed using Java with Spring Boot [27]. Spring Boot is a widely used open-source Java framework known for its stability, scalability, and large ecosystem. Spring Boot's ecosystem includes modules like Spring Security, which helps with setting up web security and authorization service and Spring Data, which simplifies database interactions. It also offers built-in support for developing structured RESTful APIs through Spring Web. All of these modules support faster development and were used in the project. In addition, Java's backward compatibility ensures long-term maintainability and reduces the risks associated with future updates.

Alternative frameworks such as Node.js [28] and Django [29] were also considered. While Node.js is a good choice for real-time and event-driven applications, it has limitations when it comes to scaling up complex CPU-heavy logic. Since the developed tool involves lots of complex natural language processing and AI-driven computations, relying on a default single-threaded runtime like Node.js would introduce unnecessary constraints. Java with Spring Boot, with its native support for multithreading and proven scalability, was therefore the better choice. Django was considered for its simplicity and strong developer experience, however due

to Python's limitations in concurrent execution and the dynamic typing model, it does not match with Java's high performance benchmarks in managing computationally demanding workflows. Therefore, the strong community backing Spring Boot, along with the author's expertise confirmed Java with Spring Boot to be the best choice for the project in terms of productivity and reliability.

4.3 Data tier

The Data tier of the application which is responsible for data storage and retrieval, was built using PostgreSQL [30]. PostgreSQL is a relational database widely known for its reliability and scalability. PostgreSQL was selected due to its strong support for complex data relationships and its ability to handle structured data. Additionally, PostgreSQL strongly follows the SQL standards and is open-source.

Alternatives such as MySQL [31] and MongoDB [32] were also considered. MySQL is a great relational database alternative but it has fewer advanced query optimization features than PostgreSQL and it is owned by Oracle, which could potentially limit its flexibility and community control in the future. MongoDB is a NoSQL database designed for schema-less data. Given the project's need to store and manage structured information such as projects, domain entities, use case specifications, test cases and their relationships, using a non-relational database would complicate rather than simplify the development process. Therefore, PostgreSQL's balance with high performance, flexibility and advanced relational data management capabilities made it a clear choice for the data tier.

4.4 AI integration

Integrating an appropriate AI model into the application was a very important architectural decision because it directly impacts the tool's quality, performance, usability and overall user experience. Given that the core AI functionality of the application is to transform unstructured

requirements into concrete and structured domain entities, use cases and test cases, the selected model had to satisfy several critical requirements:

1. **High intelligence**

   The model needed advanced natural language processing and problem-solving abilities to handle complex user inputs accurately and reliably.

2. **Accessibility and affordability**

   To preserve maintainability, the LLM chosen had to align with NFR13 by providing usage-based billing and by introducing no additional operating costs.

3. **Performance and speed**

   Fast generation of outputs was essential to ensure smooth user experience. As defined in the NFR5, generating any artifact should not take longer than one minute.

4. **Capability with large inputs**

   In order to comply with NFR14, the selected LLM had to support output volume sufficient to produce eighteen complete use cases from a single output.

An initial list of LLM candidates for the application was chosen from the LLMs leaderboard maintained by the Artificial Analysis organization [33]. The organization provides independent benchmarks and analysis of different LLM models to help users choose which AI technology fits the best for their use case. One of the most popular rankings is comparing models on a composite intelligence index. This index is derived from seven academic benchmarks (MMLU-Pro, GPQA Diamond, Humanity's Last Exam, LiveCodeBench, SciCode, AIME, and Math-500), which are very common benchmarks that AIs are ranked on. The ten highest scoring models at the time of writing the thesis that were also selected for the initial list are provided in figure 10.

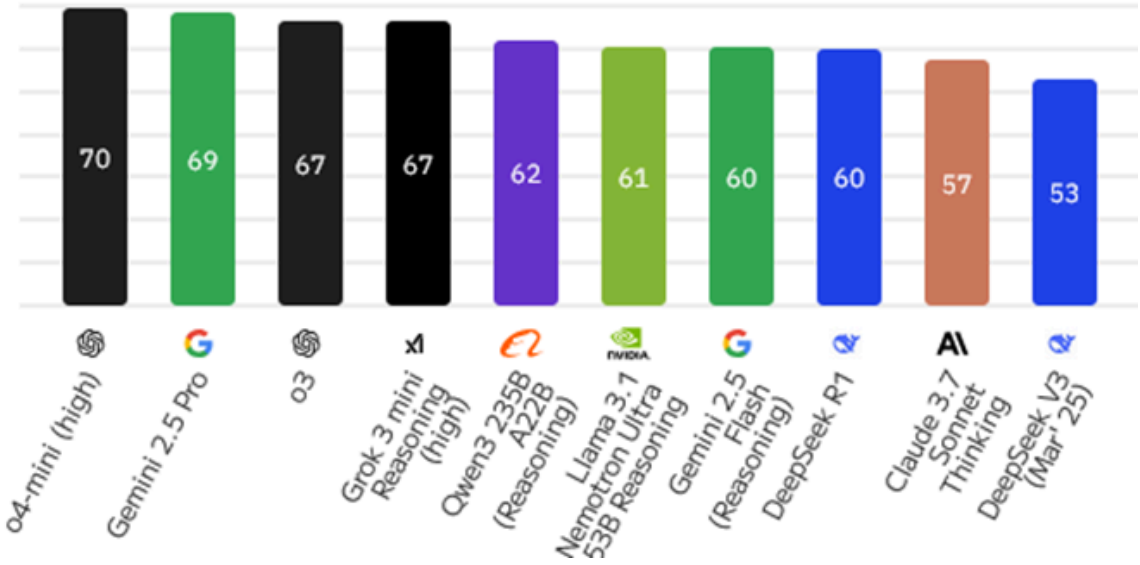


Figure 10. top ten LLM models based on the intelligence index

From the top ten most intelligent models, OpenAI's o4 mini (high) and o3 model had to be immediately ruled out due to the high cost and strict access. These models require a tier 5 subscription to OpenAI which requires a pre-payment of at least 1000 USD. Therefore, these models had to be ruled out based on the accessibility and affordability requirement (NFR13).

The next filtering factor was latency. NFR5 limits any AI-driven operations to one minute to preserve a responsive user experience and NFR14 requires that the chosen AI must be able to return 18 complete use case specifications in one output. AI measures its output in tokens and based on the Open AI documentation 100 tokens are equivalent to approximately 75 words [34]. Based on NFR15, the AI must be capable of generating use cases up to 500 words which corresponds to roughly 666 tokens, consequently eighteen of similar use cases require 11 988 tokens, meaning the LLM must generate at least 11 988 tokens per minute or roughly 200 tokens per second. Based on this requirement, the following models were eliminated:

- Llama 3.1 Nemotron Ultra 253B Reasoning
- DeepSeek R1
- DeepSeek v3

- Claude 3.7 Sonnet Thinking
- Qwen3 235B A22B (Reasoning).
- Grok 3 mini Reasoning (high)

This left two viable candidates:

- Gemini 2.5 Pro
- Gemini 2.5 Flash (Reasoning)

Both Gemini 2.5 Pro and Gemini 2.5 Flash met the defined requirements for intelligence and throughput. Both produced structured, high-quality outputs compatible with the defined use case grammar. To validate compatibility and real world performance, both models were tested using an initial prompt to generate the domain entities and their attributes, followed by a very thorough and well-structured prompt based on the input use case grammar [12] instructing the LLM to generate the use cases. Finally, following-up with a detailed prompt covering different aspects of testing such as edge cases and precondition and instructing the LLM to generate the corresponding test cases. Both models were capable of generating all 18 use cases per request with high-quality.

However Gemini 2.5 Flash demonstrated a consistent advantage in speed. The flash model generated complete outputs nearly twice as fast as Gemini 2.5 Pro with no meaningful drop in the response quality. Given the importance of responsiveness to user experience as defined in NFR5, this improvement in generation time was a decisive factor and, as a result, Gemini 2.5 Flash was selected as the final LLM for integration into the application.

5. Application overview

This chapter presents the practical outcomes of the thesis. The link to the application source code is provided in appendix 1. The web application developed was named ReqFlowly. Its name combines “requirements” and “workflow”, emphasizing its primary purpose: to automate key tasks within the requirements engineering workflow, from analyzing initial inputs to specifying use cases and generating corresponding test cases**.** Additionally, the name also aims to embody the ideal of “Requirements that flow smoothly”, highlighting the aspiration for an efficient, clear, and smooth progression throughout the requirements lifecycle. The application was designed using primarily shades of white and blue. Blue was chosen for its association with trust, stability and professionalism [35]. The following sections provide an overview of the application, describing the purpose and functionality of each component.

5.1 Components

ReqFlowly is divided into different views, including the landing page, authorization and the steps of the requirements analysis workflow from the initial requirements to the export of final artifacts. This step-by-step design ensures clarity and ease of navigation, significantly improving the user experience.

5.1.1 Landing page

The landing page is the starting point for all the users visiting the application, it is structured to provide users with an overview of the application. Starting with a header at the top of the page, which displays the application’s name and includes a login button. The landing page itself is divided into several sections. The first section, just below the header, provides users with a brief introduction to the application’s capabilities, accompanied by an illustrative mockup. Interactive action buttons labeled “Get Started” and “Learn More” direct users either to the login page or further down the landing page. An overview of the landing page as it appears to users is provided in figure 4.

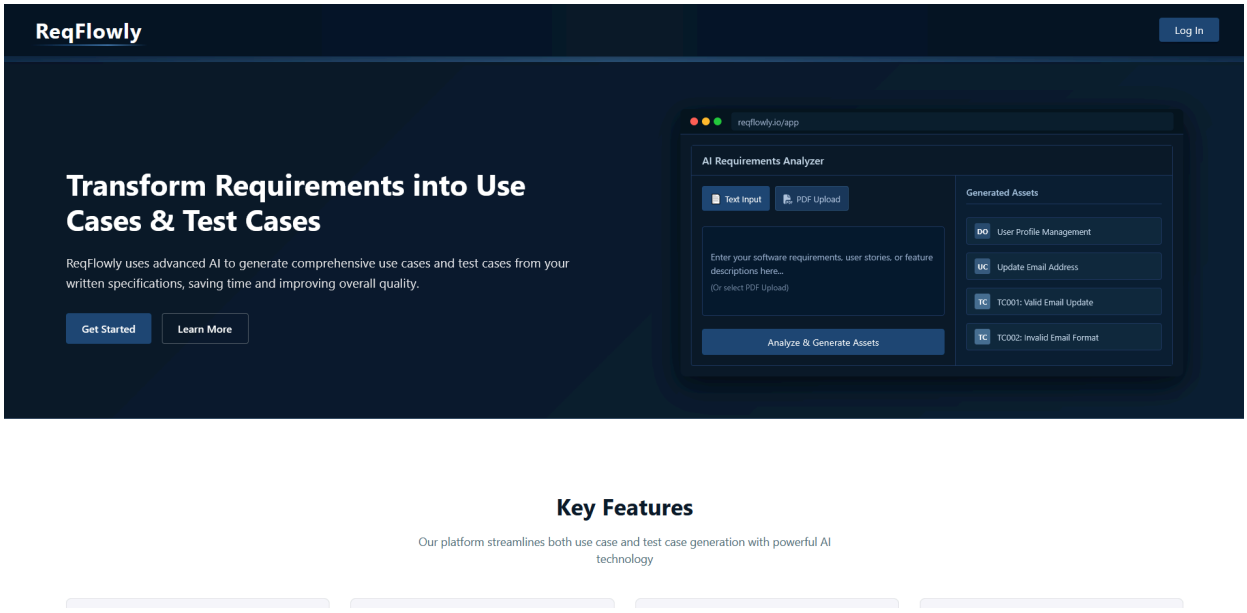


Figure 4. The landing page view scrolled to the top.

The next sections outline key features of the application and an explanation on how the tool exactly works. The landing page ends with a footer containing navigation links. The visual representation of the landing page with all its sections is provided in appendix 2. Upon clicking the “Login” or “Get Started” button, the user is directed to the login view.

5.1.2 Login view

The login view is responsible for the user authentication. It has three distinct views: create account, sign in and reset password. In the create account view, users can register an account using an email or Google account. On the sign in view, returning users can log in with previously registered credentials. The sign in view is the default view when the user clicks the login button and it is provided in the figure 5.

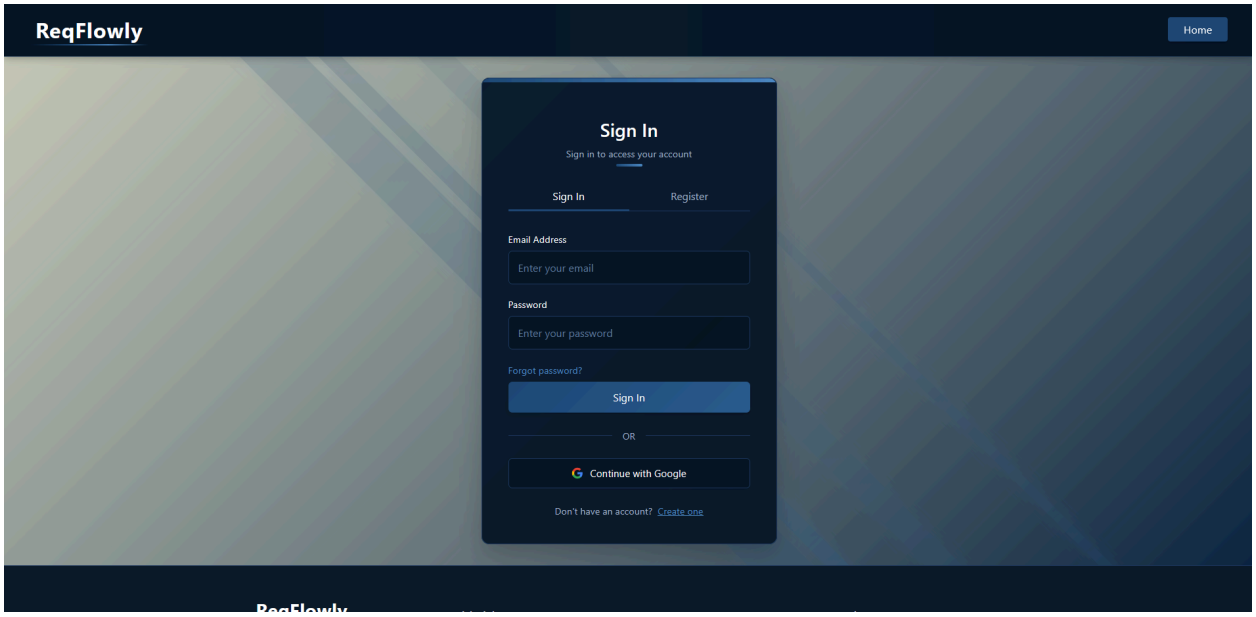


Figure 5. Sign in view of the login page.

If a password is forgotten, users can initiate a password reset via the “Forgot password?” link, which takes the user to the reset password view. The reset password link is then sent to the user's entered email address. All three login page views are provided in the appendix 3.

5.1.3 Projects view

Upon successful login, users are redirected to the projects view. In the projects view, users can either create a new project or access the previously created ones. Figure 6 shows the projects page with one created project. Users can also sort projects by name, created date and updated date. Selecting a specific project directs users to the requirements page and to the start of the requirements management workflow.

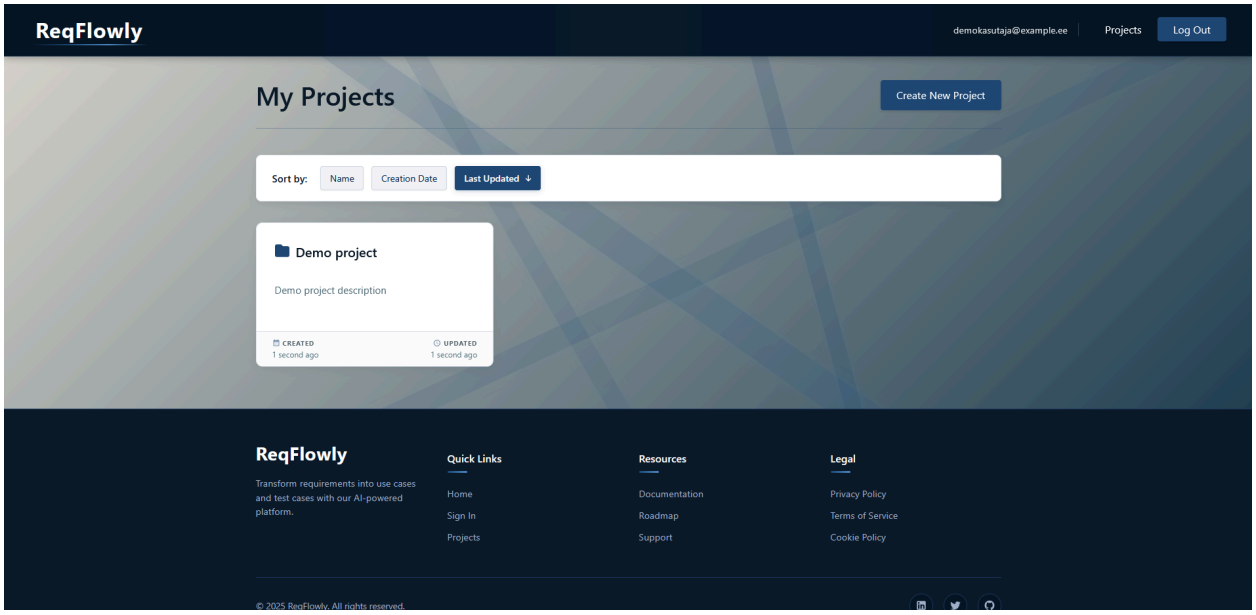


Figure 6. Projects page with one created project titled “Demo project”.

5.1.4 Requirements view

The requirements view displays the project’s metadata alongside a list of all the user added project-specific requirements. It is also the first step in the requirements management flow, which is uploading the requirements and it is displayed in the progress bar on top of the project metadata container. The purpose of this progress bar is to provide a clear, step-by-step pathway for users to follow during the requirements workflow. The requirements can be added to the project either by directly inputting text or by uploading PDF documents. Figure 7 displays the requirements page with one batch of requirements that were entered manually.

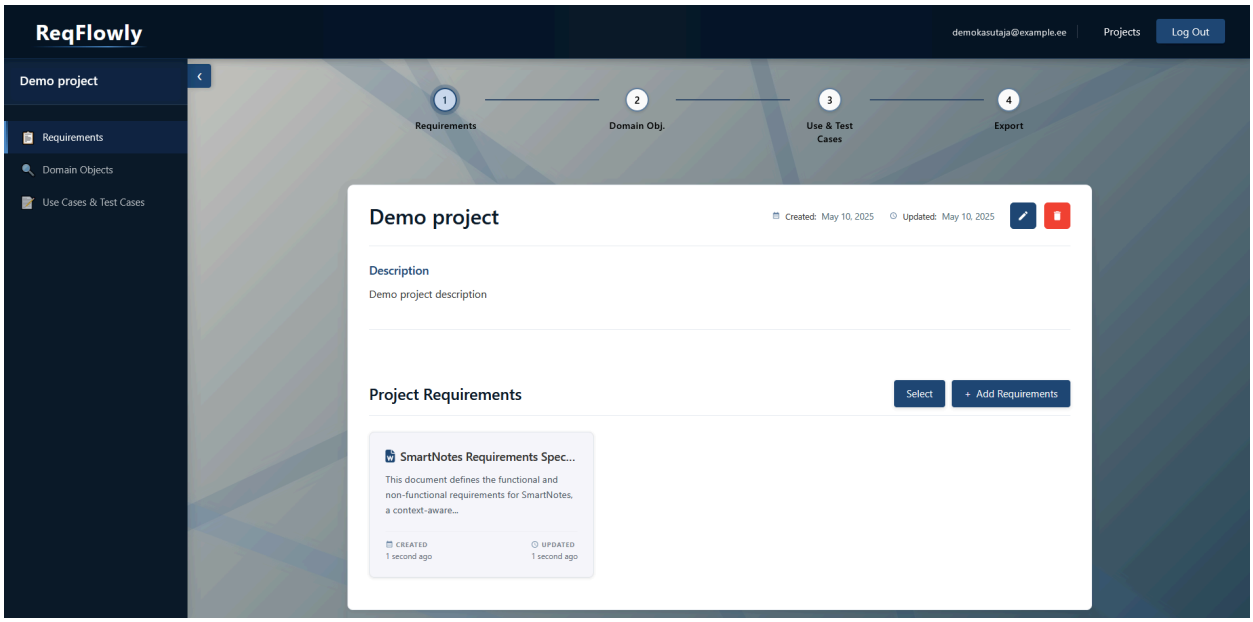


Figure 7. Requirements page with one batch of requirements.

Each added batch of requirements can be opened and individually managed, edited or deleted as needed. An open batch of requirements with managing functionalities available is provided in the figure 8.

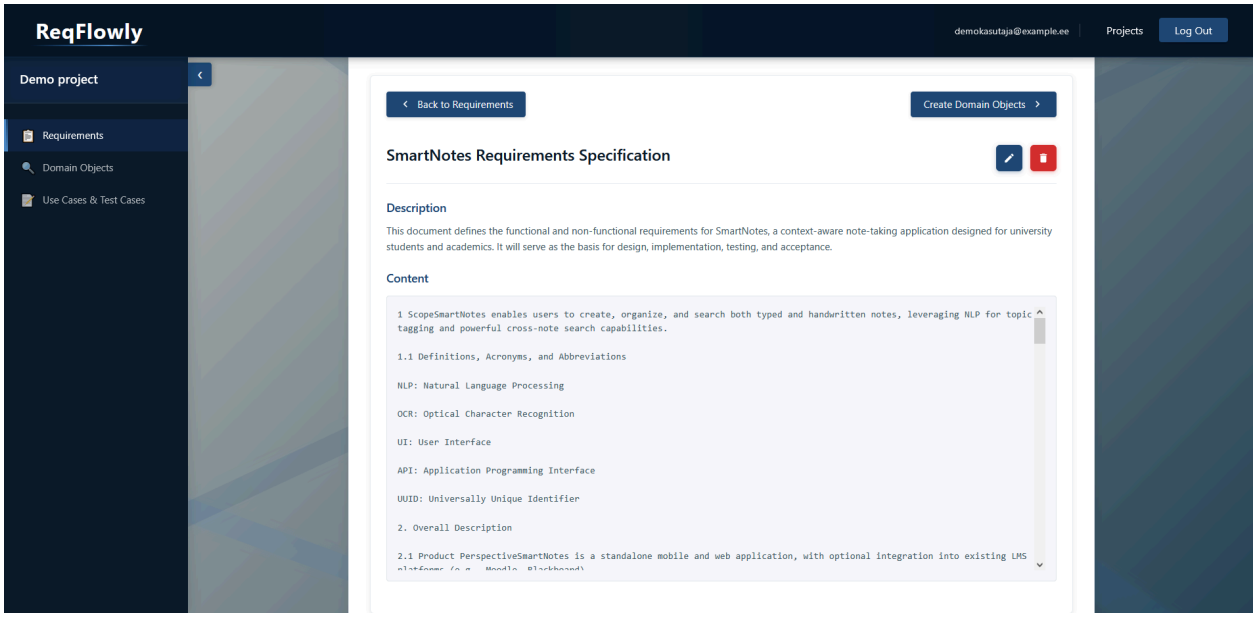


Figure 8. Open batch of requirements.

Inside the batch of requirements, users can initiate the domain object generation process by clicking on the “Create domain objects” button, which takes the user to the domain objects page.

5.1.5 Domain objects view

The domain objects view is step two of the requirements management workflow and responsible for the extraction of domain entities and their attributes from the project requirements with the help of AI. Upon selecting the requirements from the previous view, the domain objects page automatically populates relevant fields from the chosen requirements (Figure 9).

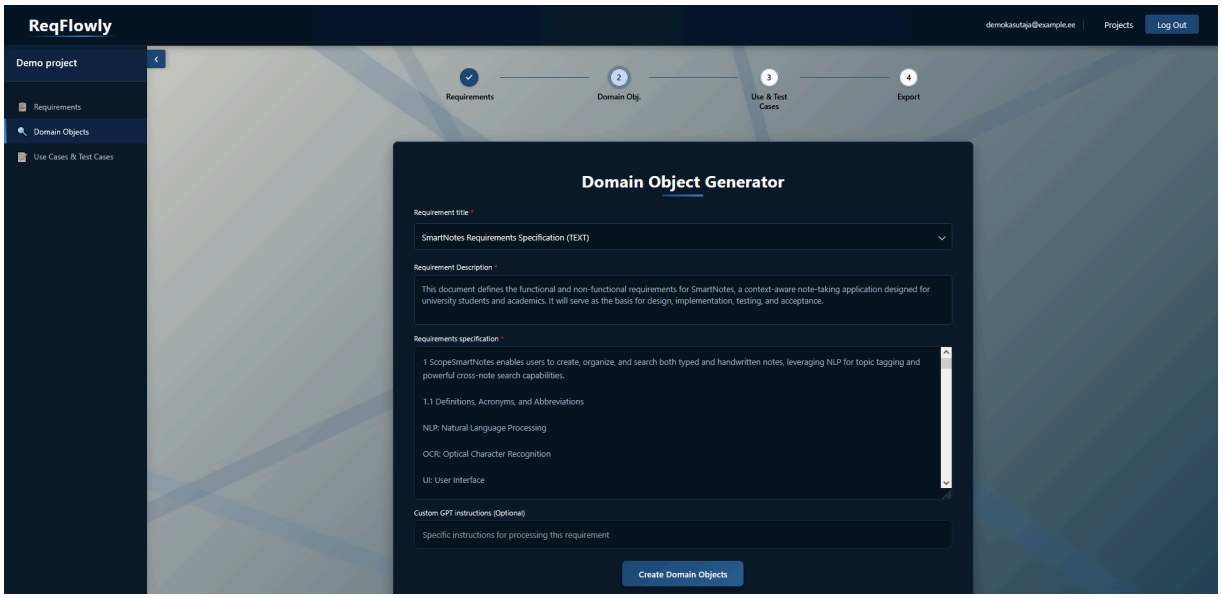


Figure 9. Prefilled domain objects view.

Users can also manually choose between all of the added requirements and tailor AI behaviour through additional custom instructions. Upon clicking “Create Domain Objects” the requirements content is sent to an AI and after some processing, the generated domain objects and attributes which are categorized into definitive and suggested items appear (Figure 10).

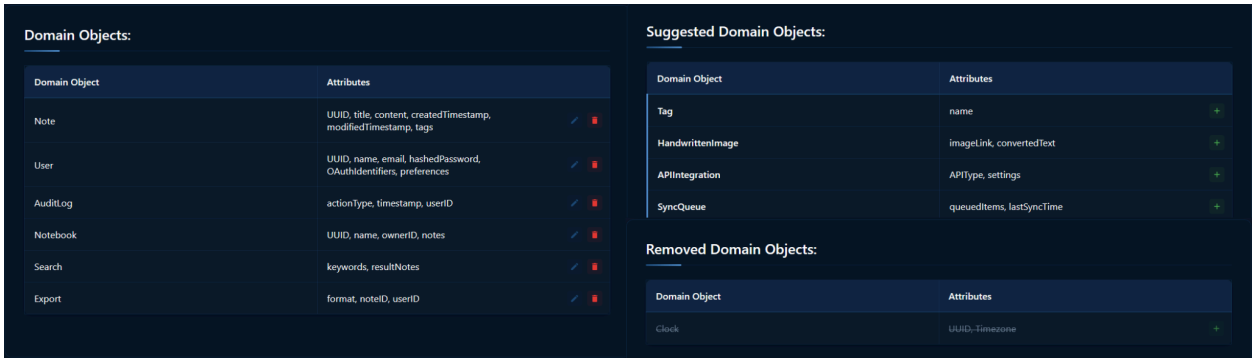


Figure 10. Generated domain objects and attributes.

Users have complete control to add, modify, or remove any of the domain objects or attributes before clicking “Save” and finalizing the domain objects from the chosen requirements. Saving domain objects directs the user to the next workflow step - use cases and test cases generation.

5.1.6 Use cases and test cases view

On the use cases and test cases view, users can manage every artifact linked to a requirement. This is step three of the requirements engineering workflow. The page opens with a table summarising the domain objects and attributes created in the previous step. Beneath the table, users can initiate AI-assisted generation of use cases by clicking the “Create Use Cases” button (Figure 11).

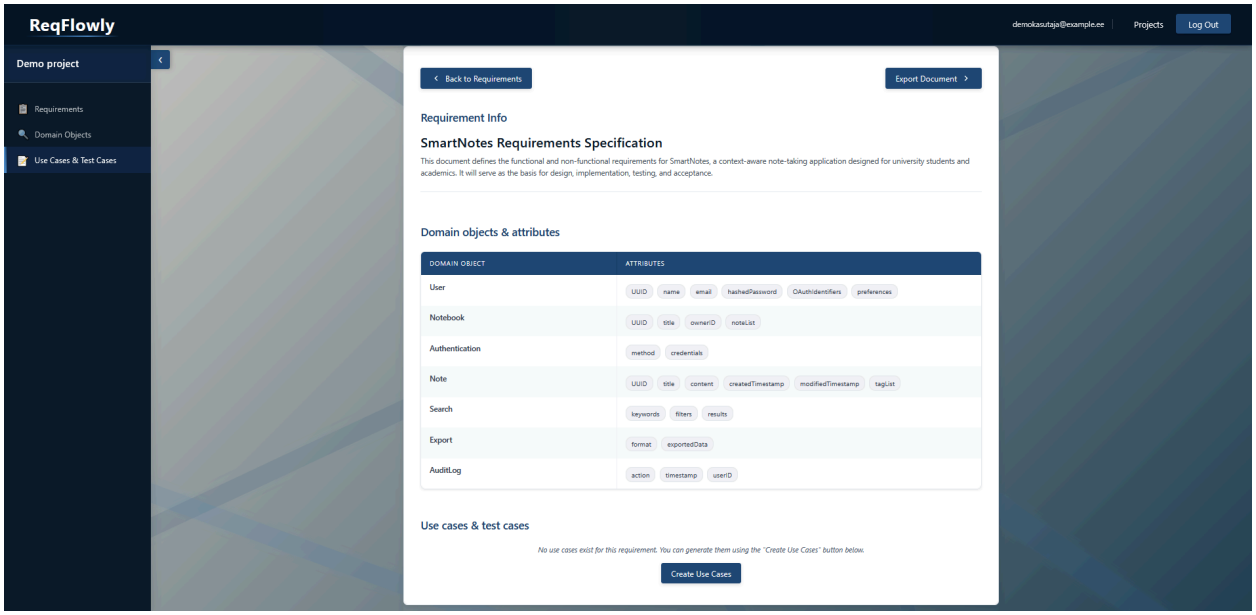


Figure 11. Use cases and test cases page.

The use cases and corresponding test cases are generated in Markdown format which allows for improved readability when displaying on the page. After creating the initial use case for a given batch of requirements, a new container is generated containing two tabs: use cases and test cases. Users can navigate between these tabs to view, edit or delete the use cases and its relevant test cases (Figure 12). Use case and test case generation views are provided in the appendix 4.

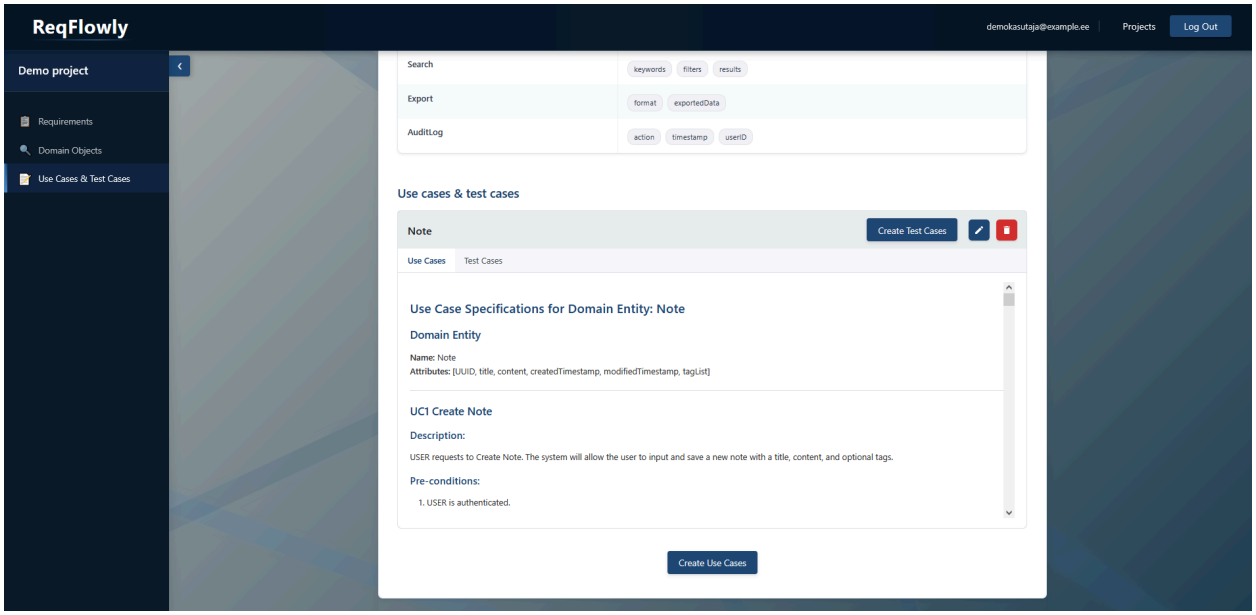


Figure 12. Use cases and test cases container.

Finally, there exists an export button in the upper-right corner of the page which allows users to navigate to the export page for the final step of requirements management workflow.

5.1.7 Export view

In the export view, users can export all the generated artifacts, including domain entities, use cases, and test cases as a structured PDF document. This is the final step of the requirements management workflow. The export interface is displayed on figure 13.

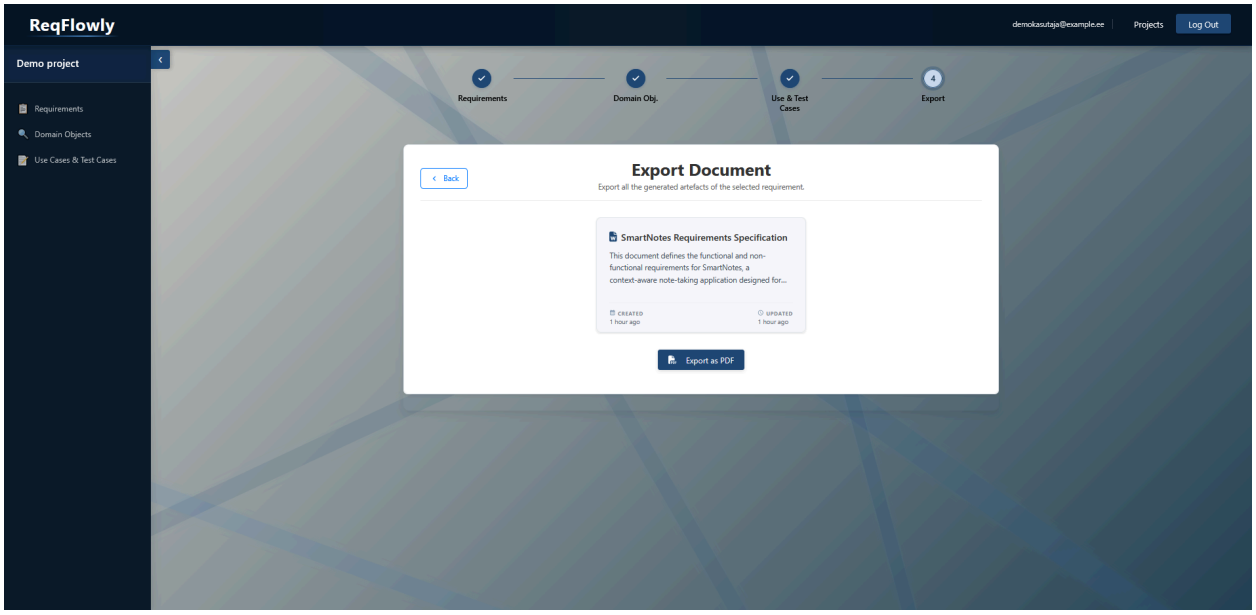


Figure 13. Export interface.

5.2 Possible further developments

Even though the current version of the application supports individual users managing small to medium-sized projects, several enhancements could significantly extend its scope and usability, particularly in collaborative and large-scale settings. Therefore, this chapter explores three possible directions for further development.

Collaborative workspace with audit trails

Implementing collaboration fatures including user roles and permissions would greatly increase the tool's usability within teams. For example, project-specific roles could include user roles such as stakeholder who is restricted to viewing and commenting, a requirements engineer with create and edit permissions and a software developer who can view and comment but also generate additional artifacts, such as pseudocode. To complement this, detailed audit trails could be implemented which logs every change in the project alongside with the author's name, timstamp and content difference. The changes would be visible in a dedicated history tab that supports side-by-side comparison views and rollback capabilities. The audit trail functionality would further reduce ambiguities and ensure compliance.

Project-level customization

Allowing project-specific customization would improve adaptability to different project needs. Users may need different AI behaviour depending on the complexity or nature of the requirements. A dedicated project settings panel could enable users to select appropriate LLM models. For example users could opt to a faster model when working with simpler requirements but choose a more sophisticated model when working with more complex data. Additionally, users could tailor prompts and add sample datasets with the corresponding artifacts they are looking to get from the data for the purpose of aligning the AI behaviour to be more close with the user's needs. Finally, general customization settings, such as language, date formats or export templates could also be added to further personalize and to improve user experience.

Extended requirements artifacts

Expanding the tool's functionalities to generate more artifacts could significantly improve the project outcomes and productivity. For example, integrating AI to analyze the initial requirements could help to identify gaps, suggest changes or convert the unstructured requirements to structured user stories. This feature would be beneficial because starting with more ready requirements also ensures higher quality domain objects, use cases and test cases. Finally, adding the capability to generate pseudocode from the finalized test cases would directly support the development phase, creating value for the added software developers in the project.

By implementing collaboration controls, project specific settings and additional artifacts generation, the application would significantly improve its capability, versatility and appeal for use by larger teams or more complex projects.

6. Results evaluation

This chapter provides the preliminary evaluation of the performance and quality of the artifacts produced by the ReqFlowly pipeline with Gemini 2.5 Flash, utilizing specifically designed prompts and the aforementioned use case grammar, against those produced by the widely recognized OpenAI model o4-mini (high), which was tested using deliberately general prompts (as described below). This OpenAI model was chosen because it is, as of the time of writing this thesis, ranked as the most intelligent AI model according to the Artificial Analysis intelligence index [33]. The comparison focuses on use case and test case generation.

6.1 Comparison of chosen LLM compared to other models

To ensure a fair comparison, both models were tasked with generating use cases for a healthcare software subsystem focusing on patient visits, examinations, and prescriptions. Additionally, generating test cases from a selected use case. Prior to the testing with OpenAI, previous chat and memory were cleared for the model to avoid bias. It was not necessary to do with the API because it has no previous memory.

6.1.1 Evaluating the use case generation

The prompt provided to OpenAI was intentionally general to reflect realistic usage:
“I am a software engineer developing the new software system of a hospital. One of the subsystems deals with patient visits, examinations, prescriptions, etc. Please give me the related use case specifications for the subsystem”

Initially, the OpenAI o4-mini (high) model generated six use cases:

- UC1: Schedule Patient Visit
- UC2: Patient Check-In
- UC3: Conduct Examination
- UC4: Prescribe Medication
- UC5: Generate Prescription Document
- UC6: View Patient Visit History

When explicitly prompted for additional prescription related use cases with the prompt “Please give me more use cases related to Prescription”, the model generated four more:

- UC7: Create Prescription
- UC8: Cancel Prescription
- UC9: Pharmacist Review Prescription
- UC10: Prescription Renewal Request

In total, the OpenAI model generated ten use cases with two prompts.

In comparison, the ReqFlowly solution, using Gemini 2.5 Flash with a specifically tailored prompt and a use of grammar generated 18 use cases in a single iteration for only one domain entity, that is, Prescription. If the tool continued to run, it would produce a similar number of use cases for each of the remaining domain entities. For the input, the used domain entity and its attributes are as follows:

Domain entity: Prescription

Attributes: prescriptionId, patientId, physicianId, medicationId, dosage, frequency, duration, routeOfAdministration, datePrescribed, refillsAllowed, refillsRemaining, status, pharmacyId, instructions, reasonForPrescription, priorAuthorization, dispenseQuantity, dateLastFilled, diagnosisId, allergyWarnings, notes, isControlledSubstance

To compare the results in more detail, the use case “Create Prescription” was selected from the outputs of both models. This use case was chosen because both of the models were able to generate it. The use case generated by OpenAI model (UC5 Generate Prescription Document) is provided in the appendix 5 and the use case generated by ReqFlowly is provided in the appendix 6.

The use case generated with the ReqFlowly application provided more and highly detailed steps. Its output meticulously breaks down interactions into specific User Actions (UA) and System Actions (SA), explicitly lists data attributes for input, and features comprehensive, multi-step alternate flows for various exceptions, making it highly formal and precise. This level of detail is

invaluable for guiding system design, development, and testing, ensuring clarity and minimizing ambiguity for technical teams by thoroughly documenting the system's behavior and handling of different scenarios. The use case generated by OpenAI on the other hand, gives only the landmarks of the process: the physician picks a drug, adjusts the dose, and confirms. It assumes the system will handle missing details automatically. As a result, the use case is more easier to understand yet it could lead to more ambiguities during implementation or testing phases.

Another notable difference is the attitude towards failures. ReqFlowly anticipates user-driven errors like incorrect medication, unsafe dosage and provides explicit correction loops and a cancellation option at every step. OpenAI o4-mini (high) by contrast does not address the user's mistakes at all and focuses solely on a rare infrastructure outage. By leaving routine validation implicit, it risks allowing dangerous prescriptions to enter production. From a patient-safety standpoint, ReqFlowly therefore offers a demonstrably stronger defence in depth strategy.

This comprehensive approach to failure management in ReqFlowly is further underscored by its significantly greater number and scope of explicitly defined alternate flows, which meticulously address a wide spectrum of common user input errors, data validation issues, and system exceptions, offering detailed recovery paths. This stands in contrast to the more limited and narrowly focused exception handling described for the OpenAI o4-mini (high) version, which primarily highlights system-level failures rather than a broad range of interactive error resolutions.

The comparison clearly demonstrates that the Gemini 2.5 Flash enhanced by the provided customized prompts and the structured use case grammar produces more superior results in terms of comprehensiveness, accuracy and usability compared to the more general response provided by OpenAI o4-mini (high).

6.1.2 Evaluating the test case generation

To ensure a fair evaluation of test case generation, test cases were generated from the “Create Prescription” use case produced by ReqFlowly. All previous context was again cleaned in the Open AI model to ensure a neutral start.

The OpenAI o4-mini (high) model received the following prompt: “Please create test cases out of this use case:" combined with the "Create Prescription" use case from ReqFlowly. The OpenAI model then generated six test cases:

- TC1: Successful Prescription Creation
- TC2: Medication Not on Formulary (AS1)
- TC3: Dosage Exceeds Safe Limits (AS2)
- TC4: Invalid Patient or Diagnosis ID
- TC5: Database Write Error (AS3)
- TC6: Cancel Creation (AS4)

ReqFlowly however produced 15 test cases. The comparison was done on the successful prescription creation test case. The create prescription test case generated by OpenAI o4-mini (high) model is provided in the appendix 7 and the test case generated by ReqFlowly is provided in the appendix 8.

The test case generated by ReqFlowly divides the happy path into twelve steps, each tied to a specific system action, formulary match, diagnosis lookup, identifier uniqueness check etc. OpenAI’s version compresses the steps into four broad actions which again could lead to more ambiguities in the software development phase.

Furthermore, ReqFlowly anchors every step to an explicit oracle, it names the exact status that must be written (for example “Pending”), the audit trail record that must be logged and the recipient list that must receive a notification. These checkpoints are concrete and data-driven and can therefore be transformed directly into automated assertions, letting software testers discover failures more efficiently. OpenAI’s border “all business-rule checks pass” expectation offers no such functionality. It forces software testers to interpret requirements on the fly, introducing human variability and raising the risk that critical regressions slip through. ReqFlowly not only describes what success looks like but also supplies the measurable criteria needed to guarantee it.

Overall, ReqFlowly demonstrated a significantly higher level of completeness, precision, and practical utility in both generated use cases and test cases. By using structured grammar, tailored prompts, and a domain modeling approach, it not only improved clarity and traceability but also introduced safeguards that directly enhanced software reliability. These results validate that the chosen model and approach deliver superior outcomes for automated requirements analysis and specification.

Conclusion

The main objective of this thesis was to design and develop a web-based application that facilitates the automation of use case specification and test case generation using LLM. The developed application, ReqFlowly, supports a complete workflow for analyzing software requirements and generating related artifacts with the goal of improving accuracy, reducing manual effort and enhancing traceability across the software development process.

The thesis introduced the theoretical foundations relevant to requirements engineering, including domain modeling, use case specification and test case design. Followed by an analysis of how AI can be integrated into the automation of these processes. The developed solution leverages AI-powered pipeline enhanced by a structured use case grammar to generate consistent, unambiguous and traceable outputs.

The implementation of ReqFlowly was implemented following a three-tier architecture to support scalability and maintainability. The user interface was developed with React, the application logic with Java and Spring Boot, and the database with PostgreSQL. The LLM selected for integration was the Gemini 2.5 Flash model, chosen for its performance, accessibility, and ability to handle the required large-scale text generation tasks.

The final application supports a fully guided workflow for requirement input, domain object extraction, use case generation, test case derivation, and export. All defined functional and non-functional requirements were successfully implemented. A comparative evaluation of ReqFlowly's outputs against those generated by the OpenAI o4-mini (high) model demonstrated that the developed tool delivers more comprehensive, structured, and testable artifacts.

The results confirm that the use of AI, combined with structured guidance and domain-driven prompts, can significantly improve the reliability and efficiency of the requirements analysis process. The tool is suitable for small to medium-sized software projects and provides a solid foundation for future improvements, including collaborative features and support for more complex project environments.

References

[1] Boehm B, Basili V. Software Defect Reduction Top-10 List. In: Boehm B, Rombach HD, Zelkowitz MV, editors. Foundations of Empirical Software Engineering [Internet]. Springer Berlin Heidelberg; 2005 [cited 2025 May 11]. p. 426–31. URL: http://link.springer.com/10.1007/3-540-27662-9_26

[2] Bajceta A, Leon M, Afzal W, Lindberg P, Bohlin M. Using NLP Tools to Detect Ambiguities in System Requirements - A Comparison Study.

[3] The Standish Group International. *The CHAOS Report* [Internet]. Boston (MA): The Standish Group; 1995 [cited 2025 May 11]. URL: https://www.standishgroup.com/sample_research/files/CHAOS1995.pdf

[4] McConnell S. An ounce of prevention. Softw IEEE. 2001 Jun 1;18:5–7.

[5] Functional vs. Non Functional Requirements [Internet]. GeeksforGeeks. 00:02:23+00:00 [cited 2025 May 12]. URL: https://www.geeksforgeeks.org/functional-vs-non-functional-requirements/

[6] Larman, C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd ed., Prentice Hall, 2005. Ch. 10 "Domain Model: Visualising Concepts".

[7] Georgiades, M. and Andreou, A. 2013. Patterns for Use Case Context and Content. In Proceedings of the 13th International Conference on Software Reuse (ICSR), (Pisa, Italy) (Rank A in 2013, Keynote speaker: Ivar Jacobson). Lecture Notes in Computer Science, Volume 7925, 267-282. Springer, Berlin, Heidelberg.

[8] I. Jacobson, M. Christerson, P. Jonsson and G. Overgaard, *Object-Oriented Software Engineering: A Use-Case Driven Approach*. Addison-Wesley, 1992.

[9] Ditty C. RML Use Case Template [Internet]. ArgonDigital | Making Technology a Strategic Advantage. [cited 2025 May 14]. URL: https://argondigital.com/resource/tools-templates/rml-use-case-template/

[10] BrowserStack, "Requirement Analysis: Definition, Importance & Best Practices," BrowserStack, Oct. 2024.

[11] Cockburn A. WRITING EFFECTIVE USE CASES. Addison-Wesley, 2000.

[12] Georgiades MG. A natural language-based methodology to formalize and automate the requirements engineering process. 2012 Jan [cited 2025 May 14]; URL: https://gnosis.library.ucy.ac.cy/handle/7/39560

[13] What Is Software Testing? | IBM [Internet]. 2021 [cited 2025 May 14]. URL: https://www.ibm.com/think/topics/software-testing

[14] Myers, G. J., Badgett, T., & Sandler, C., *The Art of Software Testing*, 3rd ed., Wiley, 2011.

[15] A. Bruch, J. Henkel and T. Zimmermann, "Large Language-Model-Assisted Software Engineering," in *Proc. Int. Conf. on Software Engineering & Knowledge Engineering*, 2023.

[16] Nebut C, Fleurey F, Le Traon Y, Jezequel J-M. Automatic test generation: a use case driven approach. IEEE Transactions on Software Engineering, Software Engineering, IEEE Transactions on, IIEEE Trans Software Eng. 2006 Mar 1;32(3):140–55. https://research-ebsco-com.ezproxy.utlib.ut.ee/linkprocessor/plink?id=ec492f55-e180-3a55-9739-6b8aa17ce523

[17] A Tool for Test Case Scenarios Generation Using Large Language Models [Internet]. [cited 2025 May 14]. Available from: https://arxiv.org/html/2406.07021v1

[18] Generative AI for Requirements Engineering: A Systematic Literature Review [Internet]. ar5iv. [cited 2025 May 14]. Available from: https://ar5iv.labs.arxiv.org/html/2409.06741

[19] Copilot4DevOps. Copilot4DevOps homepage. https://copilot4devops.com/ (08.01.2025)

[20] ScopeMaster. ScopeMaster homepage. https://www.scopemaster.com/ (08.12.2024)

[21] AgileAILabs Spec2Test product. Agileailabs homepage. https://agileailabs.com/revolutionizing-test-automation-with-spec2testai-a-leap-towards-intelligent-testing/ (08.12.2024)

[22] IBM. Three-tier architecture. https://www.ibm.com/think/topics/three-tier-architecture (19.04.2024)

[23] React homepage. https://react.dev/ (19.04.2024)

[24] *Stack Overflow. (2024). Developer Survey Results 2024. Retrieved from* https://survey.stackoverflow.co/2024/technology

[25] AngularJS. AngularJS homepage. https://angular.dev/ (19.04.2024)

[26] Vue.js. Vue.js homepage. https://vuejs.org/ (19.04.2024)

[27] Spring Boot. Spring Boot homepage. https://spring.io/projects/spring-boot (19.04.2024)

[28] Node.js. Node.js homepage. https://nodejs.org/en (14.05.2025)

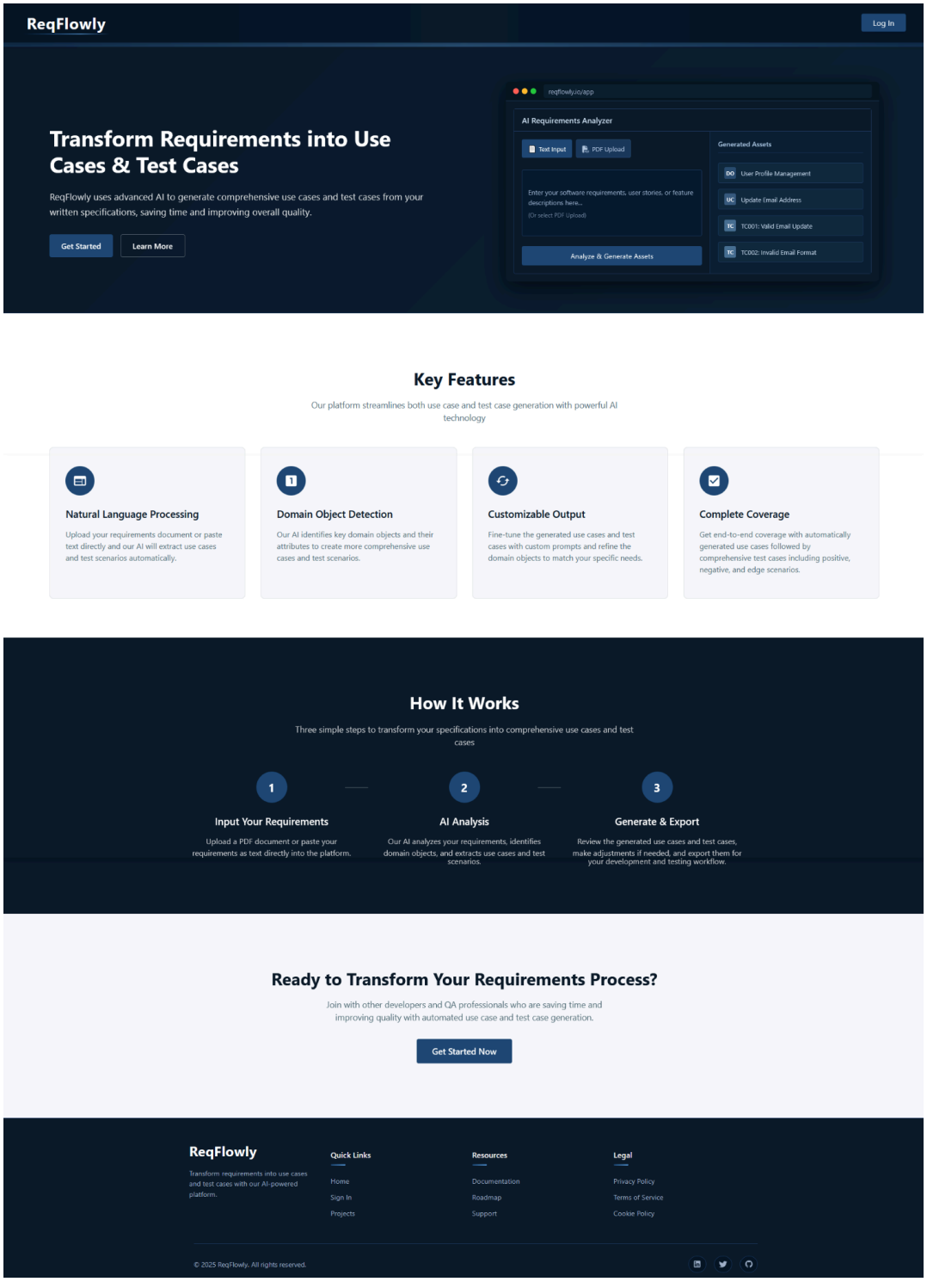
[29] Django. Django homepage. https://www.djangoproject.com/ (14.05.2025)
[30] PostgreSQL. PostgreSQL homepage. https://www.postgresql.org/ (19.04.2024)
[31] MySQL. MySQL homepage. https://www.mysql.com/ (14.05.2025)
[32] MongoDB. MongoDB homepage. https://www.mongodb.com/ (14.05.2025)
[33] Artificial Analaysis. Artificial Analysis homepage. https://artificialanalysis.ai (14.05.2025)
[34] What are tokens and how to count them? | OpenAI Help Center [Internet]. [cited 2025 May 15]. URL: https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them
[35] Figma. Blue Color: Hex Code, Palettes & Meaning [Internet]. [cited 2025 May 15]. URL: https://www.figma.com/colors/blue/

Appendices

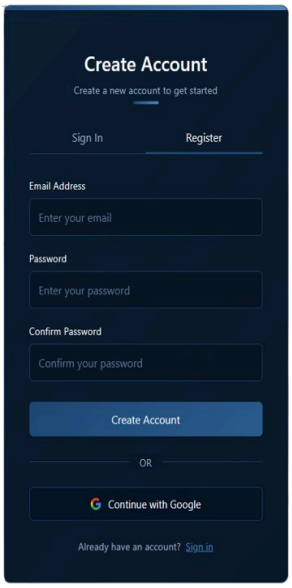
1. Source code

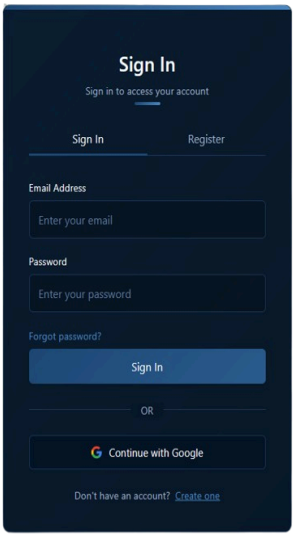
The source code for the application's front-end is available at the following GitHub repository: https://github.com/K-AMeus/reqFlowlyFE and the source code for the application's back-end is available at the following GitHub repository: https://github.com/K-AMeus/reqFlowlyBE

2. Full representation of the landing page

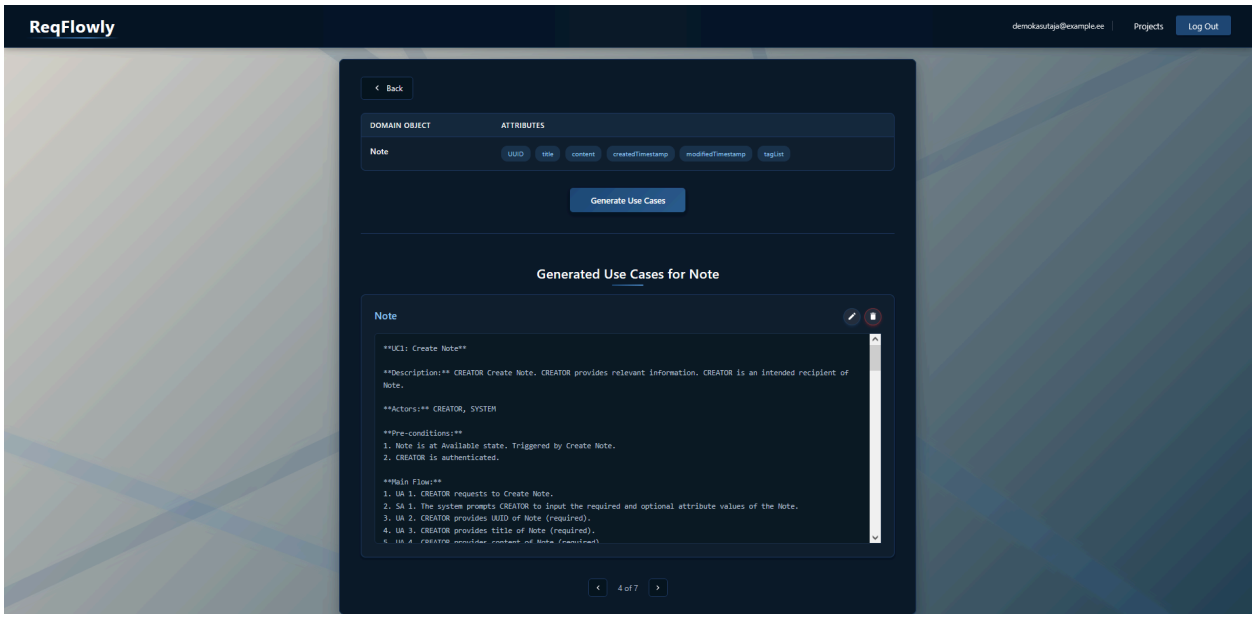


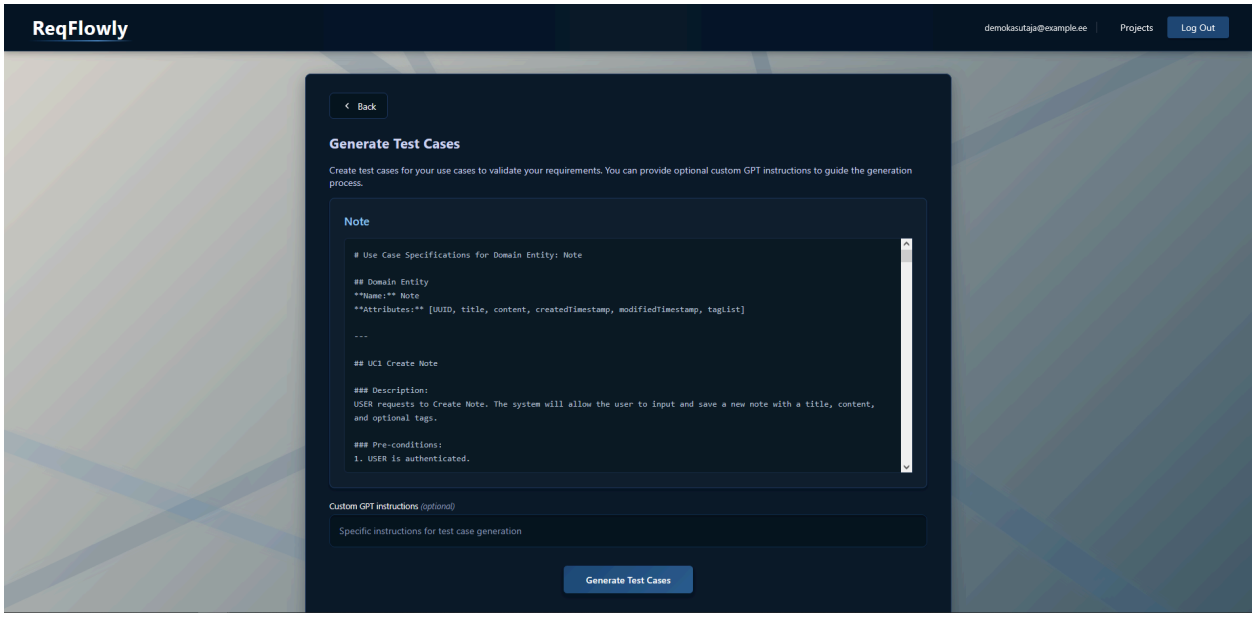
3. Views of the login page





4. Use case and test case generation views





5. Use case generated with OpenAI o4-mini (high)

Use Case ID: UC5
Use Case Name: Generate Prescription Document (Create Prescription)
Primary Actor: Physician
Stakeholders & Interests:

- Patient: needs clear prescription information.
- Pharmacist: requires accurate prescription data.

Preconditions:

- Examination (UC3) completed.

Postconditions:

- Prescription saved in system.
- Pharmacist notified of new order.

Trigger:

- Physician selects “New Prescription” in patient chart.

Main Success Scenario:

1. Physician chooses medication from formulary.
2. System auto-fills standard dosage; physician adjusts if needed.
3. Physician confirms quantity, dosage, frequency, duration.
4. System validates against allergies, interactions, and formulary.
5. System saves prescription, sends notification to pharmacy.

Alternate Flows:

- A1: Allergy or interaction detected → 2a. System warns; physician modifies choice.

Exception Flows:

- E1: Formulary service unavailable → fallback to manual entry; flag for review.

Frequency of Use: ≈80 prescriptions per day.

6. Use case generated with ReqFlowly

UC1: Create Prescription

Use Case ID: UC1
Use Case Name: Create Prescription
Description: PHYSICIAN Creates Prescription. PHARMACIST and PATIENT are intended recipients of Prescription.
Actors: PHYSICIAN (type:creator), PATIENT (type:intended recipient), PHARMACIST (type:intended recipient)
Preconditions:

- PHYSICIAN is authenticated.
- PHYSICIAN is authorized to create prescription.
- Necessary data (Patient details, Medication formulary information) for Prescription are available.

Postconditions:

- Prescription is now in Pending state.
- Prescription details are saved to the database.

Main Flow:

- UA 1. PHYSICIAN requests to Create Prescription.
- SA 2. The system prompts PHYSICIAN to input the required and optional attribute values of the Prescription.
- UA 3. PHYSICIAN provides all required and optional data for Prescription (attributes: patientId, medicationId, dosage, frequency, duration, routeOfAdministration, datePrescribed, refillsAllowed, instructions, reasonForPrescription, priorAuthorization, dispenseQuantity, diagnosisId, allergyWarnings, notes, isControlledSubstance)
- SA 4. The System validates the patientId entered previously checking existence in database.
- SA 5. The System validates the medicationId entered previously checking existence in formulary.
- UA 6. PHYSICIAN submits the data for Prescription form.
- SA 7. The System validates all submitted data for Prescription checking format compliance, data integrity, existence in database, consistency with business rules like dosage limits, formulary status, controlled substance regulations, and mandatory fields.
- SA 8. The System saves Prescription to database with reference from Prescription to associated records Patient, Medication.
- SA 9. The System confirms that the Prescription has been successfully created.
- SA 10. The System sends Notification about the creation of Prescription to the following Actors: PHARMACIST, PATIENT, PHYSICIAN.

Alternate Flows:

- Alternate Flow: AS1 Triggered by Step SA 4 Condition: Validation of patientId fails: Patient does not exist or is not accessible to the Physician.
    - SA 4.1. The System displays error message "Invalid Patient ID. Patient not found or inaccessible." to PHYSICIAN.
    - UA 4.2. PHYSICIAN corrects the patientId or cancels the operation.
    - Rejoins Main Flow at Step UA 3.
- Alternate Flow: AS2 Triggered by Step SA 5 Condition: Validation of medicationId fails: Medication does not exist in the formulary or is inactive.
    - SA 5.1. The System displays error message "Invalid Medication ID. Medication not found in formulary or is inactive." to PHYSICIAN.
    - UA 5.2. PHYSICIAN corrects the medicationId or selects a different medication.
    - Rejoins Main Flow at Step UA 3.
- Alternate Flow: AS3 Triggered by Step SA 7 Condition: Validation of submitted data fails: Missing required field, invalid format, dosage out of standard range, conflict with patient allergies, or controlled substance rules violation.
    - SA 7.1. The System displays error message listing specific validation failures to PHYSICIAN.
    - UA 7.2. PHYSICIAN corrects the data for Prescription.
    - Rejoins Main Flow at Step UA 3.
- Alternate Flow: AS4 Triggered by Step SA 8 Condition: The System database encounters a write error during save.
    - SA 8.1. The System logs the save error.
    - SA 8.2. The System displays error message "System error: Unable to save Prescription. Please try again." to PHYSICIAN.
    - The use case terminates.
- Alternate Flow: AS5 Triggered by Step UA 1..UA 6 Condition: User requests to abort/cancel the current operation. (Context: Cancellation is applicable if invoked during Main Flow steps 1..6.)
    - UA 16.1. PHYSICIAN requests to cancel the creation.
    - SA 16.2. The System prompts PHYSICIAN to confirm cancellation "Are you sure you want to cancel Prescription creation? All unsaved data will be lost.".
    - UA 16.3. PHYSICIAN confirms cancellation.
    - SA 16.4. The System discards any unsaved Prescription data.
    - . The System confirms that the Prescription creation has been cancelled. The use case terminates.

7. Test case created with OpenAI o4-mini (high)

Test Case TC1: Successful Prescription Creation
Description: Physician creates a prescription with all valid data.
Preconditions:

Physician is authenticated & authorized.

Patient, Medication and Diagnosis exist in the system.

System is operational.

Test Steps:

Physician navigates to “Create Prescription.”

System prompts for all required fields.

Physician enters valid values for all attributes:
prescriptionId=P12345
patientId=PT100
physicianId=DR200
medicationId=MD300
dosage=10mg
frequency=Once daily
duration=14 days
routeOfAdministration=Oral
refillsAllowed=2
instructions="Take with food"
reasonForPrescription="Hypertension"
priorAuthorization=false
dispenseQuantity=14
diagnosisId=DX400
allergyWarnings="None"
notes="Monitor blood pressure"
isControlledSubstance=false
Physician submits the form.

Expected Results:

System validates each ID (medication, diagnosis, patient).

All business‑rule checks pass.

Prescription is saved in Pending state.

In‑app/email notifications sent to Patient, Pharmacist, Physician.

Audit‑trail entry created.

System displays “Prescription created successfully.”

8. Test case created with ReqFlowly

A. Main Success Scenario

Test Case ID: UC1-TC-001

Title: Success – Create Prescription with Valid Data

Preconditions and Inputs:
- PHYSICIAN is authenticated and authorized.
- Patient with ID: `PAT123` exists and is accessible.
- System is operational.
- Required prescription data is available and valid:
  - prescriptionId: `RX001`
  - patientId: `PAT123`
  - physicianId: `PHY456`
  - medicationId: `MED789` (in formulary)
  - dosage: `500mg`
  - frequency: `2 times a day`
  - duration: `7 days`
  - routeOfAdministration: `Oral`
  - refillsAllowed: `1`
  - instructions: `Take after meals`
  - reasonForPrescription: `Bacterial infection`
  - priorAuthorization: `None`
  - dispenseQuantity: `14`
  - diagnosisId: `DX001`
  - allergyWarnings: `None`
  - notes: `Monitor for side effects`
  - isControlledSubstance: `false`

Test Steps:
1. PHYSICIAN requests to create a new Prescription.
2. System prompts PHYSICIAN for required and optional Prescription data.
3. PHYSICIAN provides all required and optional Prescription fields as above.
4. System validates medicationId against formulary.
5. System validates diagnosisId exists.
6. System validates patientId exists.
7. PHYSICIAN submits the Prescription data.
8. System validates input formats, integrity, and business rules.
9. System checks identifiers for uniqueness.
10. System saves Prescription.
11. System confirms successful creation.
12. System sends notification to PATIENT, PHARMACIST, PHYSICIAN.

Expected Output:
- Prescription RX001 is saved in database with status "Pending".
- Notifications are sent to all relevant actors.
- Audit trail is recorded.
- System displays confirmation message: "Prescription RX001 successfully created."

9. License

Non-exclusive licence to reproduce the thesis and make the thesis public

I, Karl-Andreas Meus

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiryof the term of copyright, my thesis Facilitating the Automation of Use Case Specifications and Test Case Generation by Developing an LLM-Powered Tool, supervised by Marinos Georgiadis

2. I grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in points 1 and 2.

4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation

Karl-Andreas Meus
15/05/2025