JOURNALS | Journal of Smart Sensors and Computing

Volume 2 Issue 4 (December 2026)
J. Smart Sens. Comput., 2026, 2(4), 26218
DOI : https://doi.org/10.64189/ssc.26218

Research Article

OPEN ACCESS

Test Case Generation Using Small Language Models: Architecture and Experimental Evaluation

Rinku Ashish Badgujar* and Prerna Pareek

School of Computer Science, Nirwan University, Jaipur, Rajasthan, 303305, India

*Corresponding Author
Rinku Ashish Badgujar
rinku.badgujar@gmail.com

Received: 22 September 2026
Revised: 03 November 2026
Accepted: 04 November 2026
Published Online: 06 November 2026

Citation

R. A. Badgujar, P. Pareek, Test case generation using small language models: Architecture and experimental evaluation, *Journal of Smart Sensors and Computing*, 2026, 2(4), 26218, https://doi.org/10.64189/ssc.26218.

Open Access

This article is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License, which permits the non-commercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as appropriate credit to the original author(s) and the source is given by providing a link to the Creative Commons License and changes need to be indicated if there are any. The images or other third-party material in this article are included in the article's Creative Commons License, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons License and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this License, visit: https://creativecommons.org/licenses/by-nc/4.0/

© The Author(s) 2026

Abstract

Designing test cases manually for banking software is still a time-consuming endeavour that frequently finds it hard to match the pace of modern agile development processes and changing regulatory requirements. Recent progress in artificial intelligence research showed some promising results in automation of various phases of the testing life cycle. However, most of such solutions use cloud-based models, require large amounts of data, or generate unstructured output, which makes them challenging to implement in an audit-proof way. In this paper, a framework for automated generation of testing scenarios and test cases based on small language models is presented. The solution comprises of a requirement extraction module (SLM-1), which extracts structured testing attributes from user stories and uses deterministic logic rules to ensure that the extracted information complies with the problem domain. The next stage involves validating the output of the requirement extraction phase, followed by the generation of a test scenario and individual steps of testing. Each generated test case in this framework will be traceable to a specific requirement label from user stories. The experiments conducted showed that the approach was able to deliver high production-level extraction performance and provided a structure and reviewable output. Results indicate that a hybrid approach using small language models, trained on particular domains, combined with deterministic test generation algorithms may serve as a feasible and implementable approach to support AI-based testing in highly regulated software domains. The research provides important insights into building enterprise-scale automated testing workflows for advanced banking applications.

Keywords: Artificial intelligence in software testing; Automated test case generation; Requirement extraction; Small language models; Banking software testing.

1. Introduction

Even though software testing is an integral component of software engineering, it is equally important in ensuring the efficient operation of software, its safety, and meeting its functional needs. When it comes to larger companies operating in sectors such as finance and banking, software testing becomes increasingly difficult, especially in light of stringent regulations and high volumes of traffic. Traditional testing approaches involve a lot of human labor to comprehend requirements, define test scenarios, and develop appropriate test cases. Such approaches are time-consuming and associated with various errors since it takes much effort to comprehend requirements written in natural language stories used in the context of agile software development. As shown in Fig. 1, Agile requirements are structured hierarchically into themes, epics, user stories, and tasks.

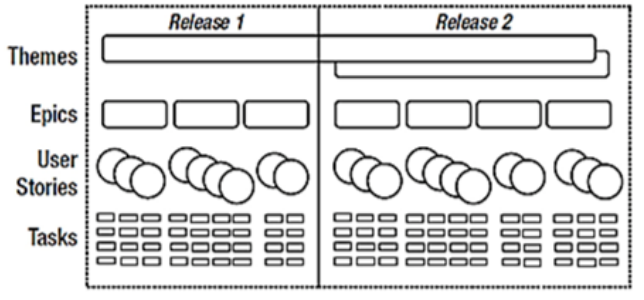


Fig. 1: Hierarchical structure of agile artifacts from themes to tasks.[1]

Fig. 1 shows how the software requirements in Agile are structured in terms of different layers such as themes, epics, user stories and tasks. This hierarchy helps to achieve traceability and makes it easy to extract test cases from the user stories through automation techniques. Hence, there is a growing interest in developing smart tools to help developers generate high-quality test cases. AI is expected to be the answer to this problem as there were quite a few successful attempts to use this technique to improve various phases of software testing, including requirements analysis, test case generation and prioritization, and defect prediction.[2-5] Moreover, surveys prove that modern machine learning and natural language processing technologies make it possible to save time during testing while increasing coverage and the level of detected defects.[6] Natural language processing, in particular, shows promise in converting informal description of software requirements into formal representations suitable for automation in testing process.[7,8] The first type of approach that has been developed by researchers is related to direct derivation of test cases from software requirements. The traditional approach involves the application of model-based techniques. However, it requires explicit modelling of the system in question in terms of a state machine or a system model. Although efficient in a controlled environment, the process requires too much effort to model a system explicitly and cannot be easily adapted to agile development settings, where the requirements change frequently.[1,9] The recent approach uses natural language processing to identify the information necessary for testing from software requirements represented in a textual format. For instance, it was proven that natural language processing could be used to create test cases that were acceptable from the perspective of requirement specification.[10] The latest trend is associated with the use of large language models (LLMs) in test case generation. A number of experiments have demonstrated that modern generative AI systems are able to automatically create unit tests and high-level test scenarios with considerable precision.[11-14] Moreover, some industry-oriented studies have investigated the use of LLMs for processing user stories and generating end-to-end testing assets, demonstrating that AI-powered testing assistants are capable of substantially decreasing the duration of preliminary test case development.[15, 16] However, despite these developments, many contemporary solutions are built around cloud-based models that might be inappropriate for use in regulated industries like finance, where data protection and infrastructure considerations often necessitate local deployments. The second problem that is encountered when using exclusively generative methods is the absence of structured and traceable output. In regulated industries, testing artifacts often need to include traceability between requirements and test cases for auditing purposes. Although generative models might be able to generate valid test cases, their connection to the requirements is often implied and cannot be easily verified. This makes many modern AI-powered testing systems impractical for enterprise-scale applications where traceability and auditing capabilities are crucial elements of QA processes.[17] These techniques intend to exploit the pattern recognition abilities of ML while ensuring that deterministic control is retained in scenario creation and result generation. Recent studies in this area have highlighted the benefit of using multiple approaches in AI-powered automated testing processes in order to increase reliability and maintainability of such testing systems, which makes this approach particularly useful in domains lacking enough training data and with high levels of regulation.[18,19] Inspired by this observation, the current research introduces a novel AI-assisted test case and test scenario generation system targeting banking software systems. The proposed system uses lightweight, on-premise deployable small language models for extracting structured attributes for testing from user stories, followed by their validation and use in generating structured test scenarios and cases within a controlled pipeline. One of the core

characteristics of this solution is its traceability, which allows each generated test case to be traced back to the requirements associated with it. The goal of the current phase of research revolves around achieving two primary objectives. The first objective relates to obtaining production-level requirement extraction results in an enterprise environment with the help of compact models, without using cloud service offerings Second, the goal is to generate structured, auditable test outputs with full traceability from requirement labelling to test generation. If these two goals are achieved, it will become possible to incorporate AI-assisted software testing systems into software development processes for banking systems without violating any operational and regulatory policies. The rest of this paper is structured as follows. Section 2 is a review of the current state of research in AI-assisted software testing, specifically in automated test generation. The proposed architecture is described in Section 3. The setup of the experiment performed to test the system is explained in Section 4. The findings of the experiments are presented in Section 5.

2. Literature review

The dynamic changes taking place in the area of software development in recent years have caused software testing to become more complicated. The testing process usually involves a lot of manual involvement while designing test cases, analyzing the requirement documentation, and checking whether software meets certain behavioral expectations or not. Though manual testing is an indispensable part of software testing, it gets hard to cope with when working on fast-evolving systems. Therefore, it was found possible to use artificial intelligence (AI) in software testing to increase the efficiency and decrease the extent of manually designing test cases.[2,6,20] As per early studies on the topic of AI in software testing, AI techniques can help to automate repetitive tasks in software testing and identify errors effectively.[2] AI-based methods can be used for prioritizing test cases, predicting errors, generating test cases, and performing other tasks related to software testing.[2,21] In addition to AI-based approaches, machine learning algorithms can contribute to decision-making processes in testing by analyzing previous data related to tests and identifying parts of software where defects are more likely to occur.[6] A third critical direction in software testing research is the development of automated test case generation methods.[22] Conventionally, automatic test generation techniques usually employ model-based test case generation in which formal models, including state machines and other system models, are employed in order to generate tests.[23] Although such techniques are able to create exhaustive tests, they demand manual effort in developing the models, making them unsuitable for agile software development processes where requirements change very rapidly. In recent times, there has been research directed at developing AI-enabled test case generation methods which would be able to generate tests based on software requirements and source codes without necessarily having to develop explicit system models.[15] Generative AI techniques have proved to be capable of creating organized test cases as well as detecting software flaws in the development cycle. [24,25] Automated unit test generation has been investigated systematically by researchers, who conclude that machine learning can be utilized in test generation.[26] NLP has also been recognized as an important technology for facilitating requirement- driven testing processes. In many software development projects, the functionalities of the systems are defined using natural language artifacts like requirement specifications, user stories, or use-case descriptions. The NLP technologies are helpful in transforming these natural language artifacts to structured data forms that can be analyzed using automation tools. As shown in Fig. 2, NLP-based pipelines analyze textual requirements using several processing

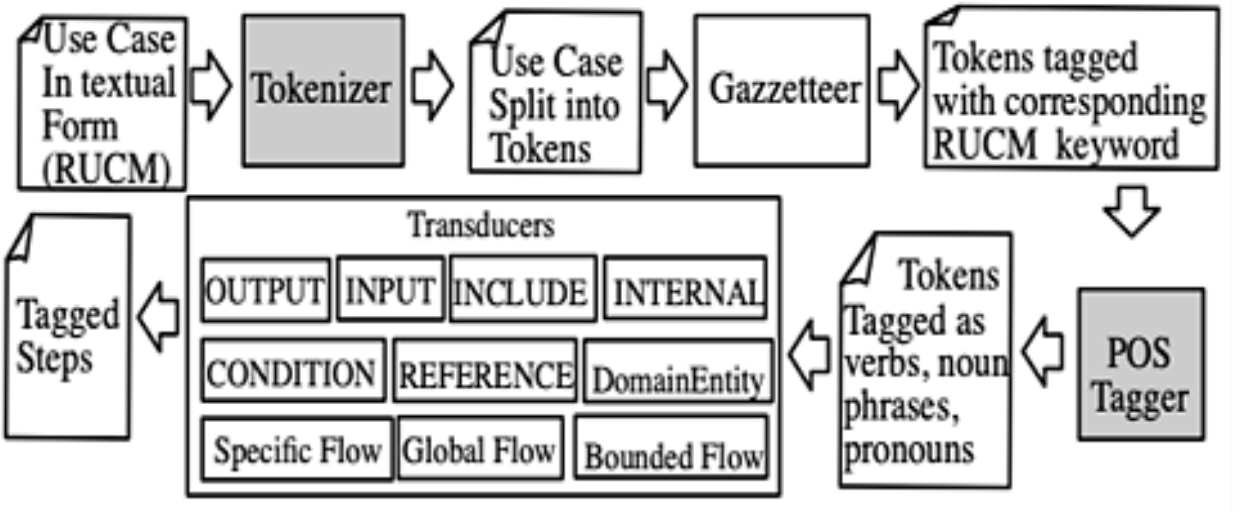


Fig. 2: NLP-based processing pipeline for extracting structured information from use case specifications.[18]

steps, including tokenization, part-of-speech tagging, and semantic analysis, to derive structured testing information.

The above diagram represents the NLP process used to transform the text-based requirement specification into a structured format. The process involves tokenization, parts-of-speech tagging, semantic analysis, and translation to domain entities. This helps in automatically extracting function elements such as actors, actions, and conditions necessary to generate test cases based on natural language requirements. Systematic literature reviews of NLP technologies in software engineering reveal that these technologies are capable of extracting functional components such as actors, system operations, and outcomes from the textual requirement specifications.[8] Therefore, automatic systems can analyze requirement specifications and infer testing information based on the extracted functional components. Several researchers have proved the applicability of NLP technologies in developing automatic test cases from requirement specifications. The studies that focus on analyzing use-case and requirement specifications have proved the feasibility of extracting acceptance test cases by identifying expected behaviors of the systems from their descriptions. The research that focuses on developing an NLP testing framework has proved the effectiveness of extracting structured testing information from requirement specifications for developing acceptance test cases.[10] The advent of large language models (LLMs) has led to rapid advancements in the field of automated test case generation. The models are able to understand intricate natural language instructions and create different kinds of software artifacts such as test cases and testing scripts. Several investigations have been carried out into the implementation of conversational AI to create test cases based on user stories or requirements.[27] The findings indicate that LLMs can create test cases that closely follow manual test case generation techniques and are therefore useful as support tools for software testing experts.[14] In the same vein, recently created automated testing solutions utilizing large language models have shown competence in handling user stories and creating test cases within IDEs.[16] Although large language models hold great promise for use in enterprise software testing, there exist multiple constraints that hinder their implementation. For instance, most available methods employ large cloud-based models that are computationally intensive and rely on an external framework. In industries such as banking and finance, which have rigorous data privacy policies, the processing of sensitive requirement documentation is usually not allowed by the external cloud-based system. Besides, the generative nature of some models leads to the creation of results whose structure is non-deterministic and lacks traceability. It is, therefore, difficult to trace how the requirements' components map to the generated test scenarios.[17] Lastly, although large language models have the ability to generate realistic testing artifacts, the output needs to be validated manually since the results may contain errors. Studies have demonstrated through extensive analysis that test cases created using such models may contain inconsistencies and fail to achieve comprehensive coverage without further validation measures.[13] As a result, several scientists have claimed that hybrid approaches that would allow machine learning models to be integrated with deterministic algorithms of testing might offer an adequate solution for testing software systems. The main goal of such hybrid AI architectures is to combine the advantages of machine learning models and deterministic approaches. The former could be used for understanding complex input data written in natural language and recognizing important features, while the latter could guarantee structured output and impose additional domain constraints. The studies of integration of various types of artificial intelligence systems in tests have shown that a mixture of particular models with the rule-based approach might be an efficient way to improve testing automation process reliability.[18] In spite of numerous successful developments in the sphere of automated test generation, there is still much to improve. For example, many existing solutions are built around generating tests rather than creating structured artifacts. This approach may result in problems related to test traceability in relation to requirements. Furthermore, some tools require the use of extensive machine learning models, which makes their deployment in on-premises enterprise settings challenging. Thus, it seems that there is a need for testing tools that incorporate both AI-based requirement interpretation and deterministic test generation processes. This research attempts to fill this gap by suggesting a mixed architecture aimed at creating test cases and scenarios for bank software applications using AI technologies. In particular, the proposed system utilizes small language models in order to extract requirements from documentation and generate structured testing artifacts based on those requirements through deterministic methods.

3. Proposed work

In this section, the architectural and methodology description of the proposed solution will be provided that should help attain the goals of the study at hand.

The most important objective that the proposed system strives to accomplish is the automation of requirement-driven test case generation for the banking software industry. The framework combines machine learning models with deterministic rules to enable correct requirement extraction and structured test

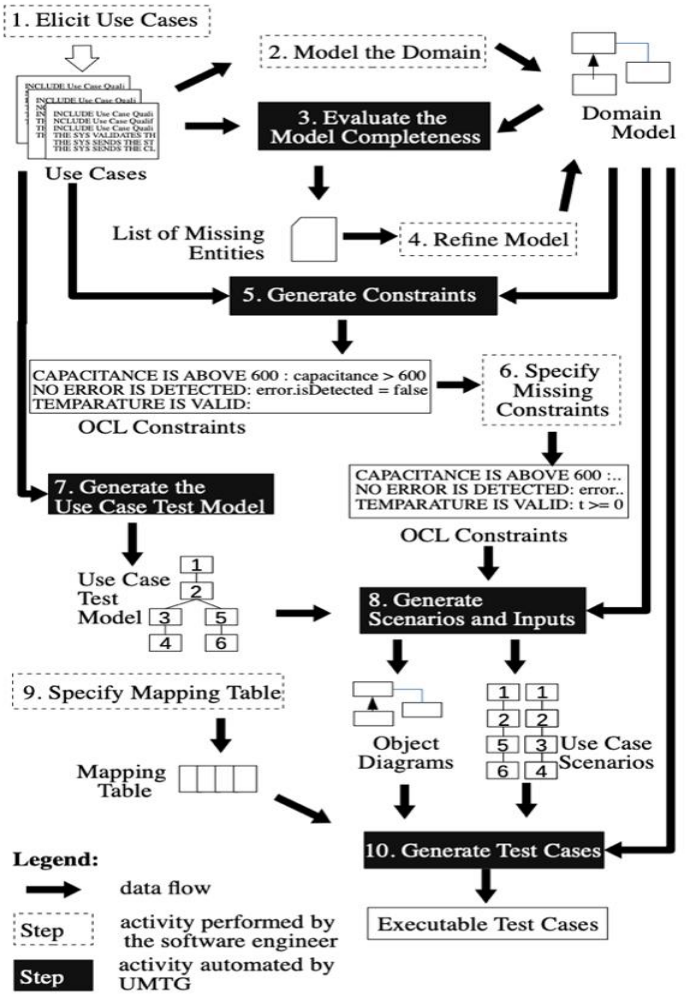


Fig. 3: Proposed hybrid architecture for requirement-driven test case generation.[10]

artifact generation. The architecture of the solution was developed based on the two goals mentioned above – requirement extraction for production environment with on-premise deployment, and generation of the structured testing artifacts traceable back to requirement labels up to individual testing scenarios. In general, the architecture comprises four main parts: requirement extraction using small language model (SLM-1), quality control/validation stage (SLM-2), deterministic scenarios generation, and test cases generation under controlled conditions using another small language model (SLM-3). These components perform their own functions within the pipeline, preserving traceability from requirements' elements all the way to the generated testing artifacts. Overall, this architecture allows translating user stories written in natural language into structured testing artifacts utilized in QA processes. As seen in Fig. 3, the suggested system uses a pipeline approach where the user stories get converted to test cases in various stages such as requirement extraction, validation, scenarios creation, and finally test case generation. The figure above is a schematic representation that demonstrates the overall architecture of the proposed system, which combines the use of small language models with deterministic rule-based modules. In the above architecture, requirement extraction is done based on user stories, and these requirements are passed through a quality gate for validation. Further, the validation of attributes is performed to develop test scenarios, which are later converted into actual test cases. Generation of tests from natural language specifications is another area that has been investigated in a number of earlier works; nevertheless, most of these methods are purely based on generation algorithms or use complicated model-based testing systems.[10] While the first option may result in realistic test inputs, it lacks determinism and traceability features required in heavily regulated industries like the financial sector.[17] It is precisely this problem that the suggested hybrid system tries to solve.

3.1 System architecture overview

The proposed framework uses a sequential pipeline in which each stage uses the output of the previous stage while preserving requirement information. The pipeline starts by extracting requirements by using a mini-

language model to detect testing attributes within user stories. Next, the obtained testing attributes go through a quality gate based on rules to guarantee consistency and completeness. Then, the labeled requirements are fed to a deterministic rule engine that generates test scenarios. Finally, test scenarios will be converted into test cases by using a structured generation process. Table 1 provides an overview of the modules considered within the framework together with their main responsibilities.

Table 1: Core components of the proposed test generation pipeline

Component	Function	Output
SLM-1 Requirement Extractor	Extracts structured testing attributes from user stories	Feature type, operation, entities, constraints
SLM-2 Quality Gate	Validates predictions and detects inconsistencies	Verified requirement labels
Scenario Generator	Generates structured testing scenarios based on requirement labels	Scenario templates
SLM-3 Test Case Generator	Produces detailed test cases with steps and expected results	Structured test cases

This kind of modular architecture enables an independent evaluation of each component with traceability between the input and output of the system preserved throughout the process.

3.2 Requirement extraction using small language models

The initial objective of this work is aimed at developing production-quality requirement extraction models capable of deployment within an enterprise setting. For this purpose, the proposed approach employs a lightweight transformer small language model designed to recognize structured testing attributes in banking software user stories. Requirement documents contain information about system operations, actors, constraints, and expected outputs that are important for test generation. Manually extracting such information would be tedious for test engineers, who might introduce variations depending on their interpretations of requirements. However, natural language processing tools can help translate requirement documents into structured formats suitable for test pipeline automation.[8] The proposed model employs a multi-classification approach and is designed to extract multiple attributes for test creation. Every classification head is responsible for extracting one specific feature of the requirement document relevant for testing. Such features have been identified through domain analysis of banking software systems and encompass various aspects of testable requirements, including system behavior, functionality, entities, and constraints. The complete set of attributes extracted by the SLM-1 model is illustrated in Table 2 below.

Table 2: Requirement feature taxonomy used for test generation

Feature Category	Description	Example
Feature Type	Functional area of the system	Payment Processing
Operation	Action performed by the user or system	Transfer Funds
Entities	System components or actors involved	Customer Account
Constraints	Conditions or limitations on system behavior	Daily transaction limit
Transition Intent	Expected system state change	Pending → Completed
Behavioral Effects	Expected outcome of the operation	Balance updated

The first step in the extraction procedure involves using the trained small language model for the tokenization and processing of the user story. Predictions are made based on the features of each category in the model, resulting in a requirement structured representation. This structured representation serves as the starting point for generating scenarios and test cases. Employing a small language model in the place of a large generative model holds certain benefits when deploying an enterprise solution. For instance, smaller models have lower computational requirements and are able to function without cloud support. Additionally, organizations gain full control over their requirement data and documents.

3.3 Requirement validation and quality gate (SLM-2)

While machine learning algorithms have been successful in extracting information in the form of requirement attributes, the problem of inaccurate predictions can arise due to ambiguities or insufficiencies in the provided requirement descriptions. As a solution to this issue, the validation layer will be included in the architecture in order to perform rule-based validation of the extracted requirement attributes. The validation process entails several verifications, such as detecting inconsistencies in the identified constraint, validating the number of entities detected, and identifying missing requirement attributes. This will ensure that the requirement representation generated is logically consistent prior to the process of scenario creation. The validation rules are summarized in Table 3.

The application of deterministic validation rules minimizes the risk of spreading erroneous interpretations of the requirements during the creation

of the scenarios. The combination of machine learning forecasts and rule-based validation techniques for hybrid systems is known to increase the reliability of automatic testing systems.[18]

Table 3: Requirement validation rules

Rule Type	Purpose
Constraint Consistency Check	Detects contradictory requirement constraints
Entity Count Validation	Ensures valid number of system entities
Domain Keyword Detection	Identifies regulatory or compliance-related terms
Confidence Threshold Validation	Flags predictions with low model confidence

3.4 Deterministic scenario generation

After the requirement attributes are validated, the pipeline generates test scenarios that represent different system behaviors. Scenarios are created based on the use of deterministic rules which create mappings between the requirement attributes and known testing templates. Each of the operation categories is matched to a certain set of test scenarios, which represent positive flows, negative cases, boundaries, and security checks among others. Use of the deterministic templates ensures that the created scenarios adhere to testing best practices. Table 4 shows some example categories of scenarios created by the system.

Table 4: Scenario categories generated by the framework

Scenario Type	Description
Positive Scenario	Valid user action producing expected result
Negative Scenario	Invalid input or constraint violation
Edge Case Scenario	Boundary conditions and limit testing
Security Scenario	Authorization and access validation
Constraint Scenario	Validation of system rules or policies

The deterministic process guarantees that the creation of scenarios is highly transparent and understandable. Every scenario can be traced back to certain attributes of requirement specification obtained during previous pipeline steps.

3.5 Structured test case generation with traceability

The fourth phase of this pipeline involves creating test cases based on the created scenarios. The proposed solution does not work like typical generator-based solutions that create free-form texts. It produces test cases, with different elements predefined (preconditions, test steps, inputs, and expected results). Test cases generated during this process remain linked to the requirement attribute elements that prompted their creation. In this way, it is possible to determine which test steps are connected to the initial requirement specification. Table 5 shows the structure of test cases generated by the solution.

Table 5: Generated test case structure

Field	Description
Requirement ID	Identifier of source requirement
Scenario ID	Identifier of generated scenario
Preconditions	System state required before execution
Test Steps	Ordered sequence of testing actions
Test Data	Input values used during execution
Expected Result	Expected system response
Traceability Label	Mapping to requirement attributes

This schema facilitates the execution of verification processes commonly necessary in enterprise environments. Traceability between requirements and test cases becomes important when dealing with regulated industries, where testing artifacts should prove compliance with the system's requirements.

3.6 End-to-end traceability mechanism

One important aspect of the developed framework is the inclusion of a traceability mechanism where the requirements attributes are traced back to test case generation. At each step in the pipeline, metadata concerning the generation process of the testing artifacts is captured. Using this metadata, the system is able to create the traceability matrix showing the mapping between the requirement attributes and the scenarios/test cases created. An example of this mapping is shown in Table 6.

Table 6: Example requirement-to-test traceability mapping

Requirement Feature	Generated Scenario	Test Case Step
Operation: Transfer Funds	Positive transaction scenario	Initiate transfer request
Constraint: Daily Limit	Constraint validation scenario	Enter amount exceeding limit
Behavioral Effect: Balance Update	Positive scenario	Verify updated account balance

This would ensure increased transparency in automated testing pipelines as well as enable the tester to ensure that the generated test cases sufficiently cover the functional requirements of the original user stories.

In conclusion, the architectural framework that has been suggested entails the combination of small language models and deterministic testing techniques to produce a

feasible platform for automated test generation for bank applications. The system will be capable of performing intelligent analysis of functional requirements and scenarios and generating test cases in an efficient and traceable manner.

4. Experimental setup

This section outlines the experimental setup employed for implementing and evaluating the framework for automating the generation of test scenarios and test cases. The purpose of this experimental setup is to examine the viability of the hybrid approach discussed in Section 3 as well as to verify its effectiveness in extracting structured requirements attributes and producing traceable testing artifacts. The implementation environment, dataset creation procedure, training procedure, evaluation process, and configuration of the entire pipeline are provided in detail to enable repeatability and transparency of the experimental procedure. This experimental setup aims to provide validation of two main research goals: reaching industrial-strength level of requirements extraction accuracy through an on-premises deployable solution and generation of traceable test cases based on requirement attributes.

4.1 Implementation environment

The proposed system was implemented as a pipeline containing machine learning blocks and rule-based modules. The implementation was carried out under the assumption that the system would run fully within the local environment to emulate an operational context similar to the one found in enterprise banking systems where cloud services could be blocked for data security reasons. Implementation was done using machine learning and natural language processing technologies in Python. Requirement extraction was implemented using transformers, which are state-of-the-art architectures for deep learning. Validation layer, scenario generator, and test case generator were implemented as rule-based blocks within the pipeline. Table 7 highlights some of the main technologies used in implementing the system.

Table 7: Implementation environment and tools

Component	Technology Used
Programming Language	Python
Machine Learning Framework	PyTorch
Transformer Library	HuggingFace Transformers
Data Processing	Pandas, NumPy
User Interface	Streamlit
Model Training Environment	CPU-based local workstation
Storage	JSON and CSV structured datasets

This combination makes it possible for the system to work even when there are no GPUs available in the environment. The architecture of the framework is therefore consistent with the goals of the research, which is aimed at designing a testing framework for deployment in enterprise environments without much overhead.

4.2 Dataset preparation

The dataset employed for training and testing the requirement extraction model was created from user stories from banking software applications. User stories refer to functional requirements that are usually expressed in agile software development frameworks and contain information on system functions, user behaviors, constraints, and outcomes of system execution. The dataset includes manually annotated user stories that were categorized based on the taxonomy of requirements outlined in the proposed framework. Labeling was accomplished through the analysis of requirement statements and the application of appropriate tags based on requirement categories like operation type, system entities, constraints, and behavior consequences. Through this process, it was possible to train a multiclass classification model that predicts several requirements attributes concurrently. The dataset configuration used in the experiments is presented in Table 8 below.

Table 8: Dataset composition

Dataset Category	Number of User Stories
Initial Training Dataset	95
Expanded Dataset	152
Target Dataset Size	200
Test Dataset	15
Extended Evaluation Dataset	20

In order to achieve a more generalized output and overcome any deficiencies concerning domain coverage encountered in the preliminary phases of evaluating models, this improved dataset was constructed. In particular, special emphasis was placed on areas that are specific to the field of banking, including transactions, account maintenance, loans, and other processes that require compliance. Requirement-based testing studies have indicated that structured datasets containing requirements are required in order to train models that can understand natural language requirements.[28] This annotated dataset allows the model to recognize typical patterns in banking software user stories.

4.3 Requirement extraction model configuration (SLM-1)

The requirement extraction component, known as SLM-1, was implemented using a lightweight transformer-based architecture that strikes the balance between efficiency and deploy ability. The model is set up as a

multi-head classifier, where a common encoder analyzes the input requirement text, and several classifiers make predictions about various requirement attributes. In this study, the adopted encoder model utilizes the DeBERTa-v3-small architecture, which ensures a high level of context representation ability without being large in size. Transformer-based models have found extensive applications in natural language processing tasks and proved to be highly efficient in capturing structured information from text data.[7] The proposed architecture comprises a common embedding and encoding stage, and separate classifiers responsible for each requirement attribute category follow. Table 9 provides an overview of the performed predictions.

Table 9: Multi-head prediction tasks for requirement extraction

Prediction Head	Task
Feature Type	Identify functional module category
Operation	Detect user or system action
Transition Intent	Identify state transitions
Entities	Extract system actors or objects
Constraints	Detect system rules or limits
Behavioral Effects	Identify expected outcomes

Each head in the classification was trained by minimizing the cross-entropy loss function on its corresponding label. The overall objective for training was the sum of losses from all classification heads.

4.4 Validation layer and quality gate implementation (SLM-2)

After extracting the labels, the predicted labels are validated through the validation layer to ensure that the logical consistency and dependability of the attribute information have been maintained. The validation layer was designed based on a rule-based approach, wherein the outputs produced by the machine learning process undergo a series of predefined tests to check for their quality. Such rules are determined based on the domain knowledge related to banking software systems and insights provided by testers. Table 10 describes the main validation methods used in the system.

Table 10: Validation rules implemented in SLM-2

Validation Rule	Description
Entity Count Check	Ensures that predicted entities fall within acceptable limits
Constraint Conflict Detection	Identifies contradictory constraint combinations
Domain Keyword Detection	Detects regulatory terms such as AML or KYC
Confidence Threshold Rule	Flags predictions with low confidence scores

Such validation rules add another layer of reliability to the machine learning algorithm preventing faulty requirement interpretation from spreading into scenario generation.

4.5 Scenario generation implementation

Once the attributes of the requirements are verified, the Scenario Generation Module is responsible for developing test scenarios that reflect various behavioral paths within the system. The module is designed to act as a deterministic rule engine that maps the requirements' attributes to the test scenarios. The deterministic nature of the Scenario Generator ensures that there are pre-defined mapping rules for each type of operation category and its corresponding scenario types. The generated scenario will have information about the requirements' attributes that spawned the particular test scenario. Table 11 summarizes the scenario templates used by the Scenario Generation Module and their corresponding testing objectives.

Table 11: Scenario generation templates

Scenario Type	Purpose
Positive Flow	Valid system interaction
Negative Flow	Invalid input handling
Edge Case	Boundary condition testing
Security Validation	Access control verification
Constraint Validation	Rule or policy enforcement

The deterministic design allows each requirement attribute to have an influence on the creation of its test scenarios. Fig. 4 below shows the Scenarios tab of the implemented interface after processing the Card Activation user story. The rule-based scenario generation module produced 15 distinct test scenarios from the extracted requirement attributes; each assigned a unique identifier following the SC-US-CARD-001-XXX naming convention. Scenarios are color-coded according to their priority. HIGH scenarios marked in red cover the critical positive flow and the key business rule constraint violations while MEDIUM and LOW scenarios in orange and green address validation checks, edge cases and entity-level verifications. Each entry on the right-hand side carries a traceability tag referencing the specific SLM-1 output field that triggered its creation, for example slm1.operation. positive for the successful activation scenario and slm1.constraint. expired card blocked for the corresponding business rule scenario. This direct tagging between scenario and source attribute is what ensures the traceability guarantee described in the framework design, giving testers immediate visibility into why each scenario exists.

4.6 Test case generation module (SLM-3)

4.6.1 Generation of test cases from scenarios

The fourth and last module in the pipeline develops detailed test cases based on the scenarios developed

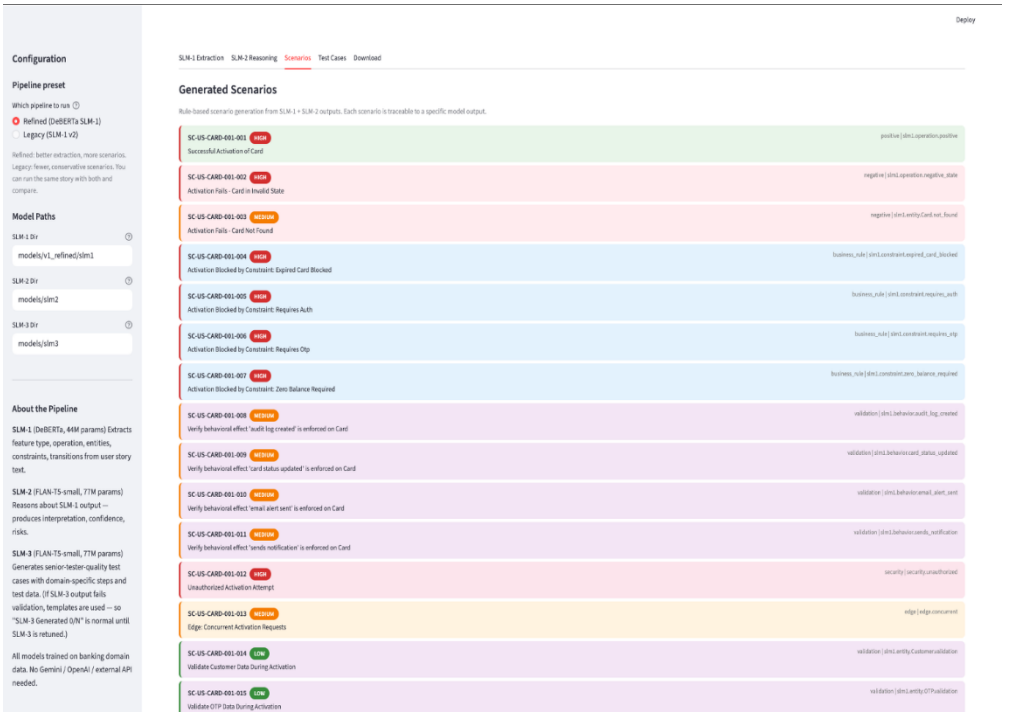


Fig. 4: Generated Scenarios view within the Streamlit interface displaying rule-based scenarios for the Card Activation user story, categorised by type and priority with traceability tags to SLM-1 attributes.

earlier in the process. The module uses structured templates and text generation techniques to generate test cases that can be used during the process of software testing. Each test case developed by the tool is composed of elements used during software testing. These include preconditions, actions, input values, and output results. Table 12 summarizes the structure of the generated test cases and the information maintained for traceability and execution purposes.

Table 12: Generated test case format

Field	Description
Test Case ID	Unique identifier
Scenario Reference	Scenario that generated the test case
Preconditions	Required system state
Test Steps	Ordered sequence of actions
Test Data	Input values
Expected Result	Expected system behavior
Traceability Link	Reference to requirement attributes

The use of such a structure guarantees the readability of generated test cases. Fig. 5 illustrates the Test Cases tab of the pipeline interface, showing the expanded view for the first generated test case i.e. TC-SC-US-CARD-001-001 which covers the successful activation of a card. The test case is assigned a priority of P0 and a severity of critical reflecting its status as the primary positive flow scenario. The preconditions section specifies that the card must exist in the system, be in a valid state for activation and the user must be authenticated and authorized before the test begins. The test steps follow a clear, ordered sequence from logging in and navigating to card management, through initiating and completing the activation request with OTP verification, to confirming the status update and checking that a notification has been dispatched to the relevant parties. The expected result is stated plainly and the test data block below provides concrete identifiers for card, customer and OTP to be used during execution. The coverage and source metadata at the bottom of each card traces the test case back to its originating scenario and the SLM-1 field that produced it keeping the full chain of traceability intact from raw requirement through to executable test step.

4.7 End-to-End pipeline execution

For the assessment of the entire framework, the experimental arrangement consists of the implementation of a whole process flow covering execution of all phases within the end-to-end execution pipeline where user stories would be processed by the entire set of components developed within the framework. These steps, as illustrated in Fig. 6, include:

1. Input user story parsing and ingestion
2. Extraction of requirement attributes via SLM-1
3. Validating those attributes via SLM-2
4. Scenarios creation via deterministic templates
5. Test cases creation via structured templates
6. Traceability mapping for the generated test artifacts

The above diagram presents an overview of the execution process of the framework that was adopted in the experiment. The process starts from receiving the input

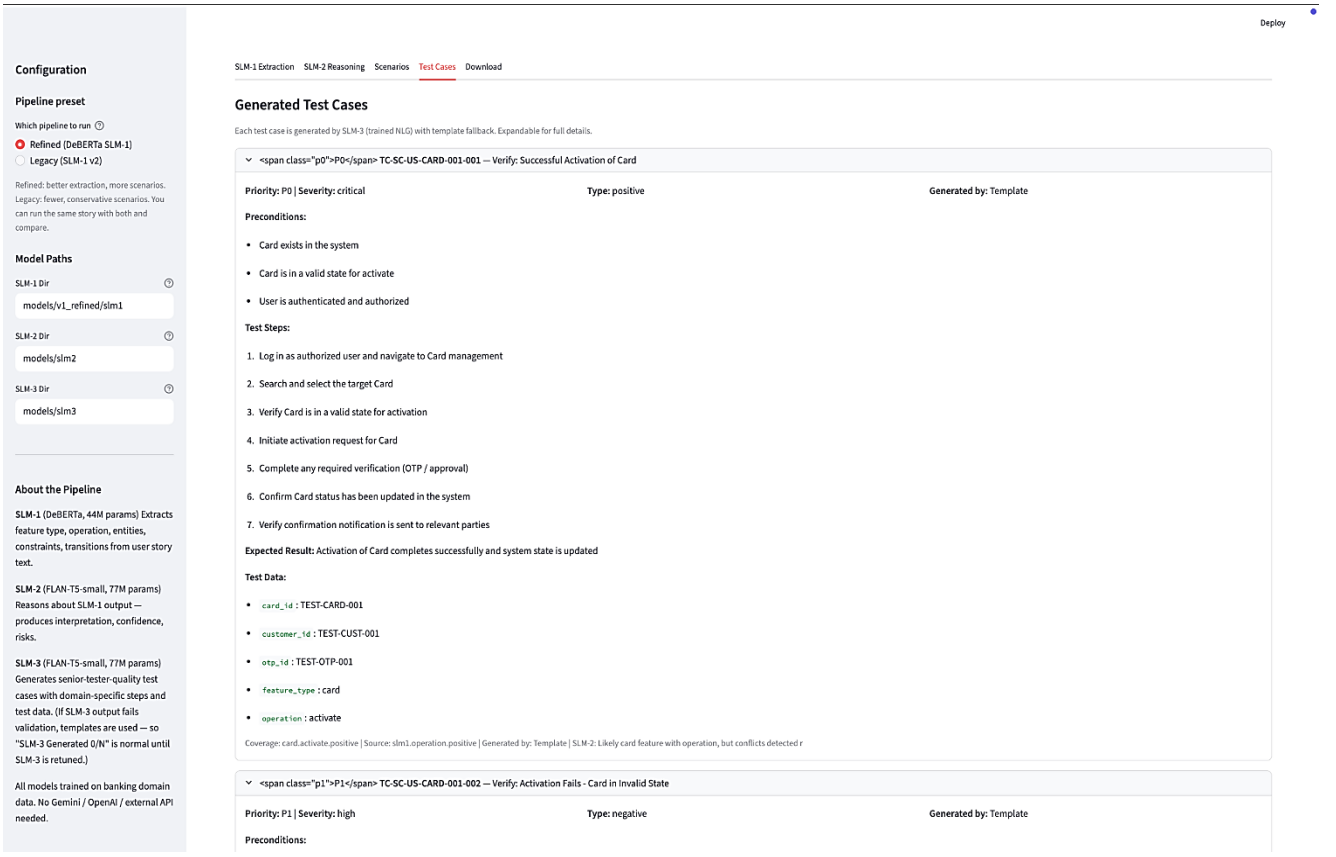


Fig. 5: Generated test cases view in the framework interface showing a fully expanded test case for the successful Card Activation scenario, including preconditions, ordered test steps, expected result, and test data.

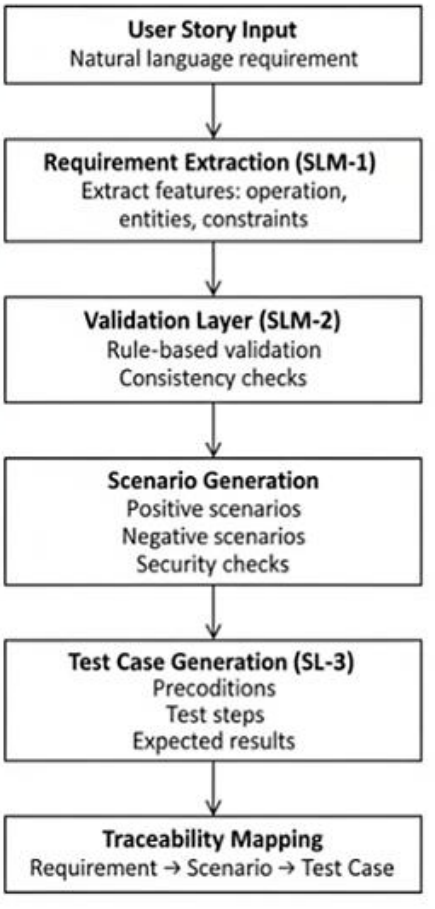


Fig. 6: End-to-end execution pipeline for automated test case generation.

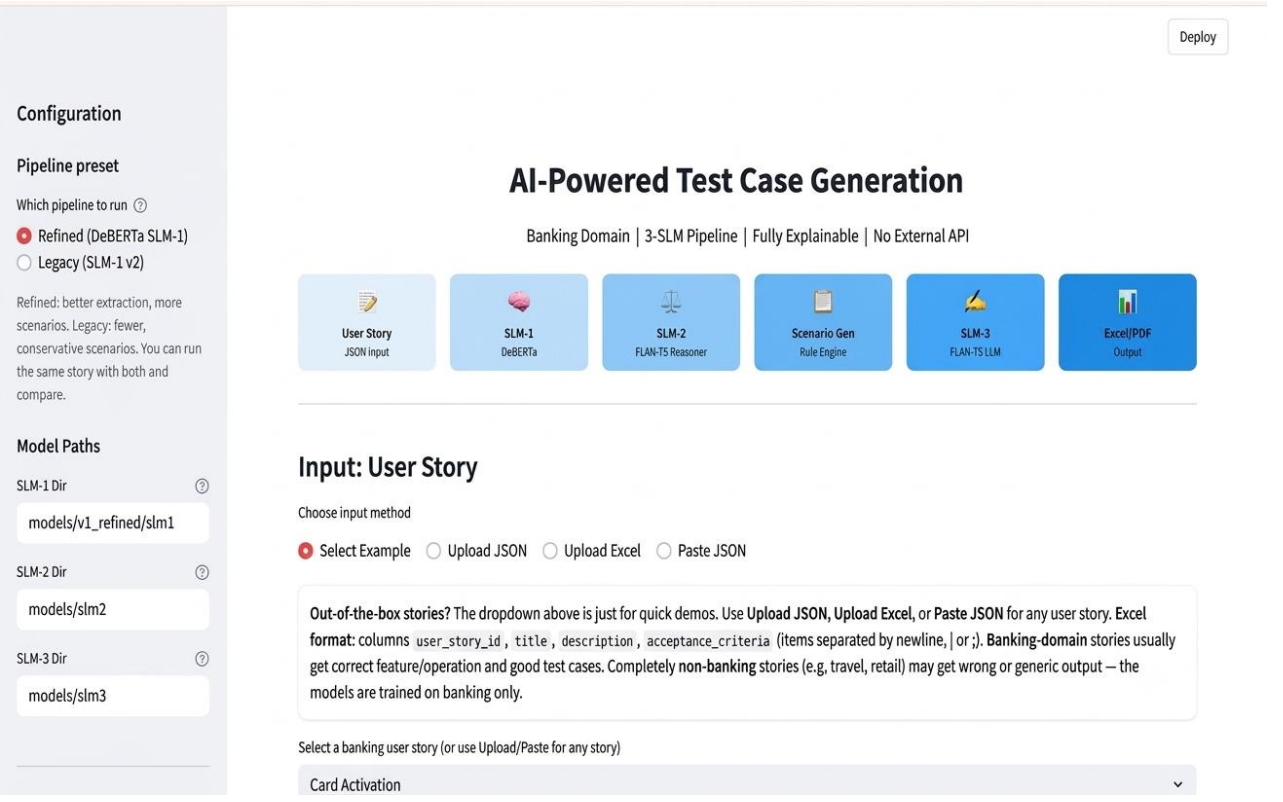


Fig. 7: Implementation interface of proposed multi-SLM test case Generation system

data based on user stories and proceeds to extracting requirements (SLM-1), validating requirements (SLM-2), generating deterministic scenarios, and structured test cases. The last step guarantees the traceability among requirements, scenarios, and created test cases.

Fig. 7 illustrates the implemented interface for the proposed framework. The interface is built on Streamlet so that the full end-to-end pipeline can be run in a local deployment environment. The paths for the three SLM models that do requirement extraction, validation and test case generation can be configured dynamically by the users. The interface also visualizes the sequential workflow of ingestion of user stories, extraction of SLM-1, validation of SLM-2, generation of scenarios and

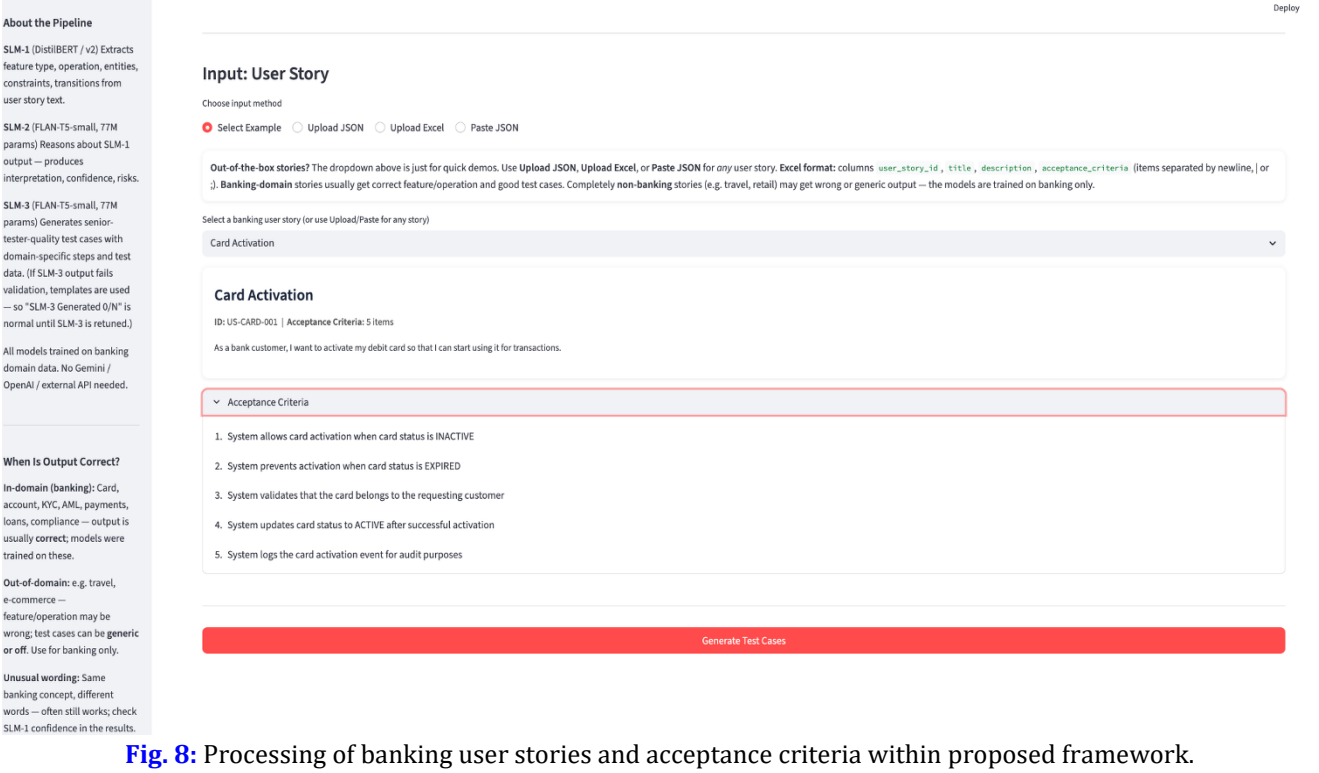


Fig. 8: Processing of banking user stories and acceptance criteria within proposed framework.

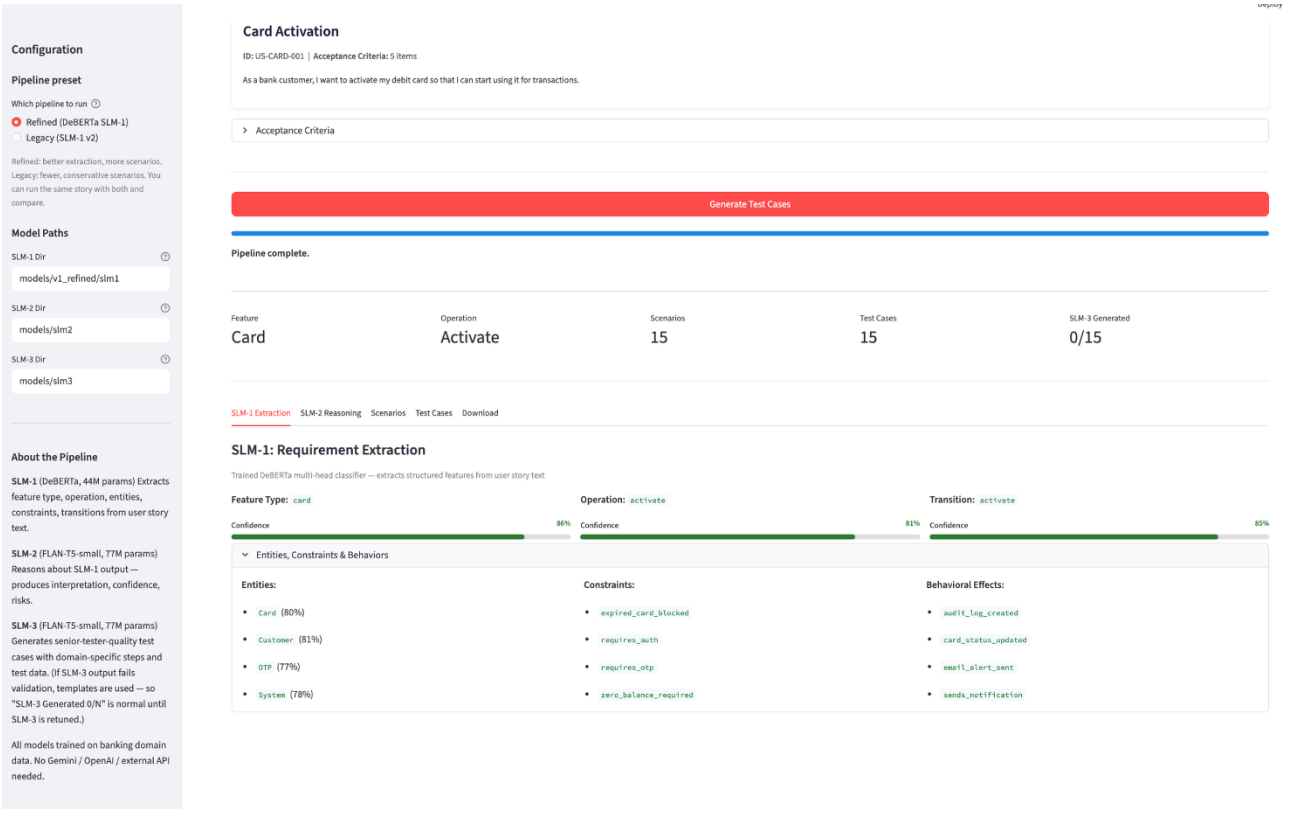


Fig. 9: SLM-1 requirement extraction output for card activation user story showing extracted feature type, operation, transition, entities, constraints, and behavioural effects with associated confidence scores.

creation of SLM-3 based test cases. Supports JSON and Excel input, improving usability for enterprise testing workflows. The fully local deployment architecture fits the banking environments and the goal of an explainable, on-premises framework. For the evaluation purpose, the pipeline was tested via a set of genuine user stories for banking applications which contain such scenarios as account handling, transaction processing, and other issues associated with regulatory compliance.

Fig. 8 shows the processing of a banking-domain user story in the implemented framework. The example demonstrates a structured input to the pipeline of a card activation requirement and its acceptance criteria. The interface allows the user to see the functional requirements and validation constraints before generating test scenarios and test cases. The acceptance criteria shown are for various behavioral conditions including card state validation, ownership validation, status transition handling and audit logging requirements. The implementation presented demonstrates the ability of the proposed framework to deal with realistic banking requirements and translate them into structured testing artefacts through the integrated multi-SLM architecture.

Fig. 9 captures the SLM-1 extraction stage output for the Card Activation user story within the implemented pipeline. Once the user story is submitted and pipeline execution completes, the interface presents a consolidated summary on the first card, confirming the extracted feature as Card, the operation as Activate and reporting a total of 15 scenarios and 15 test cases generated from a single input. The SLM-1 tab breaks this down further by displaying the confidence readings for feature type (86%), operation (81%) and transition intent (85%) all of which fall comfortably above the confidence threshold defined in the validation layer. The lower section of the panel lists the identified entities like Card, Customer, OTP and System alongside the detected constraints such as expired card blocked, requires auth, requires OTP and zero balance required and the expected behavioral effects including audit log creation, card status update, email alert and notification dispatch. Seeing all of these extracted attributes together in one view makes it straightforward for a tester to verify that the model has correctly understood the scope and intent of the requirement before any scenarios or test cases are produced. Table 13 below shows the user stories used for testing of the end-to-end framework.

The current experiment configuration allows for an assessment of the entire pipeline and provides empirical data about the possibility of generation of structured test cases from natural language input via automated processes. In general terms, it is safe to state that the experimental arrangement proves the potential applicability of the developed hybrid architecture implemented within the local environment.

5. Results and discussion

Table 13: Banking user stories used for pipeline evaluation

Story Category	Example Functionality
Card Management	Card blocking and activation
Compliance	KYC verification
Account Management	Account closure
Lending	Loan eligibility validation
Payments	Transaction processing

The following describes the experimental results that have been extracted through an evaluation of the automatic test generation process described in this paper. In particular, this evaluation has been conducted by analyzing the efficiency of the process with respect to the main goals of this work – i.e., precise requirement extraction through a light and implementable model and generation of valid test cases appropriate for use within an enterprise testing environment. The results have been analyzed according to three major axes.

5.1 Requirement extraction performance

In the initial phase of the evaluation, we aimed at evaluating the performance of the requirement extraction model (SLM-1). The evaluation was conducted based on two sets of data, including the original testing set and an extended evaluation set that incorporated new user stories that describe complex banking processes. Evaluation was carried out on several prediction heads for different requirements' attributes, which include feature type, operation categorization, transition intention, entities, constraints, and behavior effect. Single label classification problems were evaluated with accuracy metrics, while multi-label prediction problems were evaluated with macro-averaged F1 scores. The accuracy metric was computed as the ratio of correctly predicted labels to the total number of evaluated instances. For multi-label prediction tasks, the macro-averaged F1 score was used, representing the harmonic mean of precision and recall across all labels. The corresponding formulations are given in Equations (1) and (2). Table 14 summarizes the requirement extraction performance of SLM-1 across different prediction heads for both evaluation datasets.

$$Accuracy = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \quad (1)$$

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (2)$$

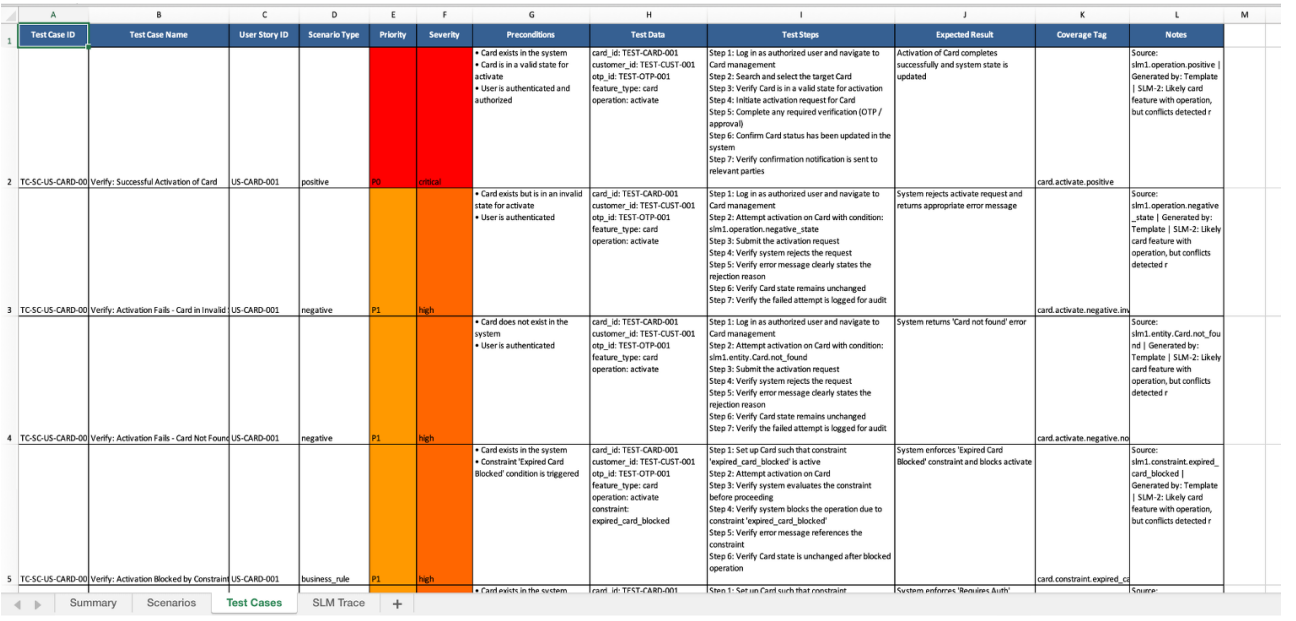
As it may be seen, the proposed approach shows high-performance levels in the vast majority of prediction problems, especially concerning feature categories and operations. As was expected, a slight drop in the model's performance could be observed in terms of working with a larger and more diverse set of requirements. However, despite this fact, the accuracy level of the model stayed stable in most of the classification heads, which means that the lightweight architecture used by the researchers is capable of processing such difficult inputs. It should be pointed out that expanding the dataset helped to increase the performance levels in the case of constraint and behavioral effect prediction, which were characterized by poor accuracy before since they had low levels of representation in the training sample. Such results may be explained by the fact that the expanded training dataset helps the system to recognize patterns of constraints that are often used in the case of banking software systems and represent some limits on possible transactions. Thus, it may be concluded that small language models can reach competitive levels of performance in requirement extraction while being computationally light enough to use on-premises.

Table 14: SLM-1 Requirement extraction performance

Prediction Head	Metric	Test Dataset (n=15)	Extended Dataset (n=20)
Feature Type	Accuracy	1.00	0.95
Operation	Accuracy	0.93	0.90
Transition Intent	Accuracy	0.88	0.86
Entities	F1 Score	0.91	0.88
Constraints	F1 Score	0.74	0.79
Behavioral Effects	F1 Score	0.71	0.76

5.2 Test case generation quality

The next phase of the evaluation aimed to assess the quality of test cases generated by the SLM-3 test generation module. The evaluation procedure involved a set of held-out scenario descriptions which had not been seen by the training dataset. For each scenario description, the test case generated by SLM-3 was compared to a manually developed test case made by a professional QA engineer. In order to compare the generated test case and the reference one, we utilized some automated metrics measuring the similarity of their textual representations. More specifically, we used BLEU-4 and ROUGE-L scores, which usually are employed in research on natural language generation. Although such metrics are widely utilized in NLP studies, they are only partially applicable for test case quality assessment, as there may be several correct test cases corresponding to one scenario description. To ensure more objective



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Test Case ID	Test Case Name	User Story ID	Scenario Type	Priority	Severity	Preconditions	Test Data	Test Steps	Expected Result	Coverage Tag	Notes	
2	TC-SC-US-CARD-00	Verify: Successful Activation of Card	US-CARD-001	positive	P0	critical	• Card exists in the system • Card is in a valid state for activate • User is authenticated and authorized	card_id: TEST-CARD-001 customer_id: TEST-CUST-001 otp_id: TEST-OTP-001 feature_type: card operation: activate	Step 1: Log in as authorized user and navigate to Card management Step 2: Search and select the target Card Step 3: Verify Card is in a valid state for activation Step 4: Initiate activation request for Card Step 5: Complete any required verification (OTP / approval) Step 6: Confirm Card status has been updated in the system Step 7: Verify confirmation notification is sent to relevant parties	Activation of Card completes successfully and system state is updated	card.activate.positive	Source: slm1.operation.positive | Generated by: Template | SLM-2: Likely card feature with operation, but conflicts detected r	
3	TC-SC-US-CARD-00	Verify: Activation Fails - Card in Invalid	US-CARD-001	negative	P1	high	• Card exists but is in an invalid state for activate • User is authenticated	card_id: TEST-CARD-001 customer_id: TEST-CUST-001 otp_id: TEST-OTP-001 feature_type: card operation: activate	Step 1: Log in as authorized user and navigate to Card management Step 2: Attempt activation on Card with condition: slm1.operation.negative_state Step 3: Submit the activation request Step 4: Verify system rejects the request Step 5: Verify error message clearly states the rejection reason Step 6: Verify Card state remains unchanged Step 7: Verify the failed attempt is logged for audit	System rejects activate request and returns appropriate error message	card.activate.negative.inv	Source: slm1.operation.negative_state | Generated by: Template | SLM-2: Likely card feature with operation, but conflicts detected r	
4	TC-SC-US-CARD-00	Verify: Activation Fails - Card Not Found	US-CARD-001	negative	P1	high	• Card does not exist in the system • User is authenticated	card_id: TEST-CARD-001 customer_id: TEST-CUST-001 otp_id: TEST-OTP-001 feature_type: card operation: activate	Step 1: Log in as authorized user and navigate to Card management Step 2: Attempt activation on Card with condition: slm1.entity.Card.not_found Step 3: Submit the activation request Step 4: Verify system rejects the request Step 5: Verify error message clearly states the rejection reason Step 6: Verify Card state remains unchanged Step 7: Verify the failed attempt is logged for audit	System returns 'Card not found' error	card.activate.negative.no	Source: slm1.entity.Card.not_found | Generated by: Template | SLM-2: Likely card feature with operation, but conflicts detected r	
5	TC-SC-US-CARD-00	Verify: Activation Blocked by Constraint	US-CARD-001	business_rule	P1	high	• Card exists in the system • Constraint 'Expired Card Blocked' condition is triggered	card_id: TEST-CARD-001 customer_id: TEST-CUST-001 otp_id: TEST-OTP-001 feature_type: card operation: activate constraint: expired_card_blocked	Step 1: Set up Card such that constraint 'expired_card_blocked' is active Step 2: Attempt activation on Card Step 3: Verify system evaluates the constraint before proceeding Step 4: Verify system blocks the operation due to constraint 'expired_card_blocked' Step 5: Verify error message references the constraint Step 6: Verify Card state is unchanged after blocked operation	System enforces 'Expired Card Blocked' constraint and blocks activate	card.constraint.expired_ca	Source: slm1.constraint.expired_card_blocked | Generated by: Template | SLM-2: Likely card feature with operation, but conflicts detected r	
6							• Card exists in the system	card_id: TEST-CARD-001	Step 1: Set up Card such that constraint	System enforces 'Requires Auth'		Source:	

Summary Scenarios Test Cases SLM Trace +

Fig. 10: Test Cases sheet from the exported Excel workbook showing structured test cases with preconditions, ordered test steps, test data, expected results, and coverage tags for the Card Activation user story.

evaluation, a domain expert conducted manual analysis of generated test cases based on three qualitative criteria: step completion, correctness of banking algorithms, and clarity of expected results. Table 15 summarizes the outcomes of our evaluation.

Table 15: SLM-3 test case quality evaluation

Scenario Type	BLEU-4	ROUGE-L	Expert Score (/5)
Positive Scenarios	0.52	0.65	4.2
Negative Scenarios	0.44	0.58	3.9
Security Scenarios	0.38	0.52	3.6
Constraint Scenarios	0.41	0.57	4.0
Edge Case Scenarios	0.33	0.48	3.4
Overall Average	0.41	0.58	3.82

Fig. 10 shows a portion of the Test Cases sheet within the exported Excel workbook produced by the pipeline for the Card Activation user story. Each row corresponds to a generated test case and captures the full set of fields defined in the framework's test case template scenario type, priority, severity, preconditions, test data with specific card and customer identifiers, a numbered sequence of test steps, the expected system response and a coverage tag linking back to the originating scenario. The color coding on the Priority and Severity columns provides immediate visual triage, with P0 critical cases highlighted in red and P1 high-priority cases in orange allowing QA teams to quickly identify which test cases need to be executed first in a regression cycle. The Notes column on the far right records the generation source and the SLM-2 interpretation for each case, making it straightforward to identify any test cases that may warrant closer human review before being added to a test suite. The structured, row-based format of the output is designed to slot directly into standard test management workflows without requiring additional reformatting. It is found from the evaluation that the created test cases exhibit a high degree of correlation with the reference test cases written by hand. The positive scenarios received the highest evaluation score owing to their simplicity in terms of expected behavior and overall structure. The negative and constraint scenarios showed a good performance level due to their successful generation during the deterministic scenario's creation phase. The edge case scenarios had somewhat reduced similarity scores owing to the complexity involved in generating edge case scenarios. However, according to the experts' assessment, the created test cases can be used for the initial version of tests. This finding is supported by earlier research indicating that the test cases generated by AI are effective in creating the first version of the test cases.[13]

5.3 End-to-end pipeline evaluation

For assessing the practical usefulness of the suggested framework, an end-to-end evaluation was performed based on real-world banking user stories. The user stories went through the entire process flow involving requirement extraction, requirement validation, scenario development, and test case development. For end-to-end testing purposes, five sample banking scenarios were chosen. The details are listed in Table 16.

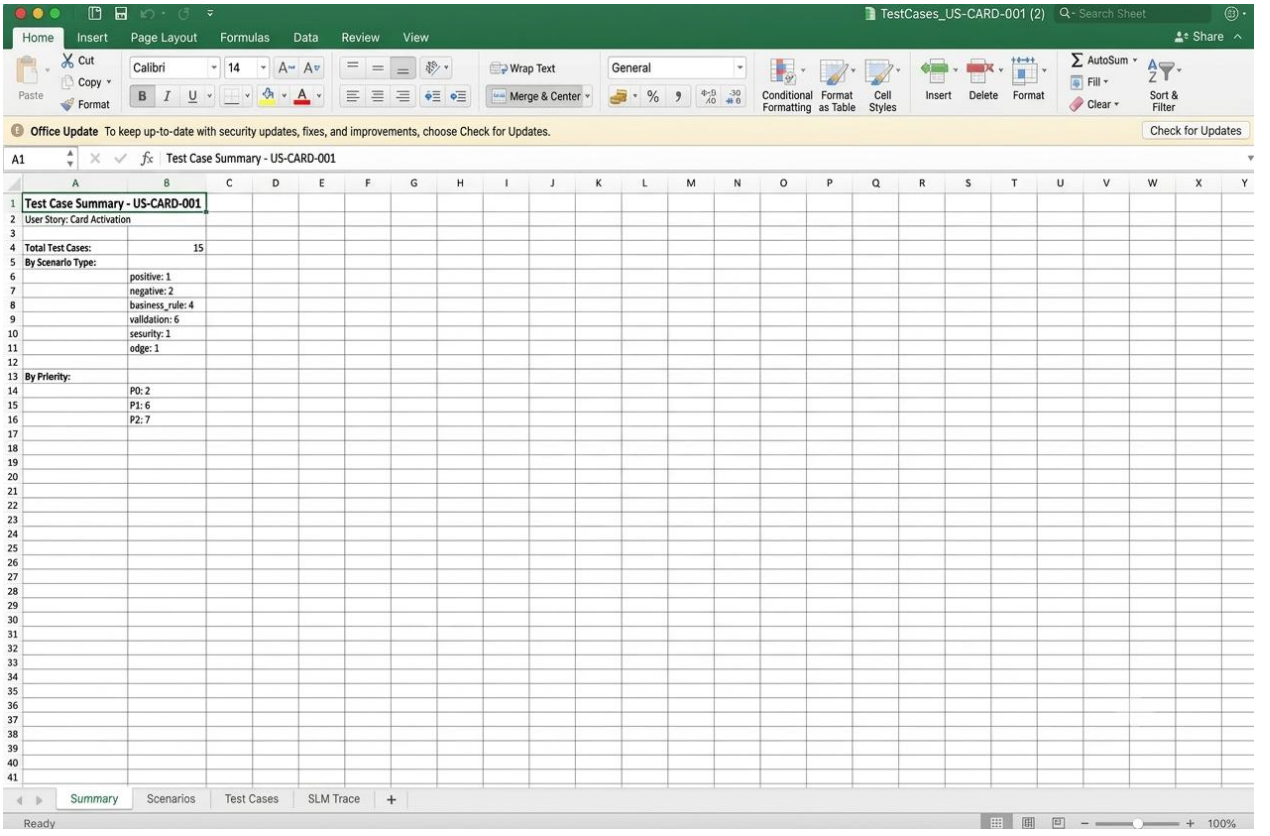


Fig. 11: Summary sheet of the exported Excel output for the Card Activation user story, showing total test case count and breakdown by scenario type and priority level.

Table 16: User stories used in end-to-end evaluation

Use Case	Description
Card Blocking	Customer blocks a lost debit card
KYC Verification	System validates customer identity
Account Closure	Customer requests account termination
Loan Eligibility	System evaluates loan approval criteria
Payment Processing	Customer initiates fund transfer

Each user story came with test cases that were then evaluated against those test cases written manually by a QA engineer. The emphasis of such comparison was on assessing the coverage of test scenarios as well as their usefulness. The outcomes of the evaluation are presented in Table 17 below.

Table 17: End-to-end pipeline evaluation results

Metric	Manual Testing	Proposed System
Average Scenarios per Story	9.2	10.4
Scenario Coverage	—	89%
Generation Time	~6 minutes	1.8 seconds
Expert Quality Rating	—	Acceptable with minor review

Fig. 11 presents the Summary sheet of the Excel file generated by the pipeline for the US-CARD-001 Card Activation user story. The sheet provides a concise statistical breakdown of the output confirming that a total of 15 test cases were produced from the single input story. Broken down by scenario type the distribution reflects the multi-path coverage strategy of the framework: 4 business rule cases, 6 validation cases, 2 negative cases, 1 positive case, 1 security case and 1 edge case. Viewed alongside the priority breakdown 2 at P0, 6 at P1 and 7 at P2 this distribution shows that the pipeline weights its output appropriately, ensuring that the most critical functional paths receive the highest-priority test coverage. These summary statistics offer a quick way for QA managers to assess the overall shape of the generated test suite before committing it to a test management tool and they also serve as a useful input for reporting on test coverage at the requirement level.

The findings show that the proposed approach was able to develop slightly higher numbers of test scenarios than conventional manual methods of testing in terms of user stories. In addition, the test scenarios generated through this system had an average test scenario coverage of 89% against the manually created test scenarios. This shows that the approach effectively captures all the critical test paths suggested by human testers. One other major benefit that emerged from the evaluations is the decreased test case development time.



1	**SLM-1: Requirement Extraction**			
2	**Field**	**Value**	**Confidence**	**Top Candidates**
3	Feature Type	card	0.863	card (0.86), account (0.66), kyc_onboarding (0.03)
4	Operation	actIvate	0.809	activate (0.81), update (0.08), freeze (0.03)
5	Transition Intent	activate	0.850	activate (0.81), block (0.04), none (0.03)
6				
7	**Entities Detected**			
8	Card	conf: 0.802		
9	Customer	conf: 0.808		
10	OTP	conf: 0.771		
11	System	conf: 0.778		
12				
13	**Constraints Detected**			
14	expired_card_blocked	conf: 0.821		
15	requires_auth	conf: 0.751		
16	requires_otp	conf: 0.719		
17	zero_balance_required	conf: 0.721		
18				
19	**Behavioral Effects**			
20	audit_log_Created	conf: 0.878		
21	card_status_updated	conf: 0.730		
22	email_alert_sent	conf: 0.719		
23	sends_notification	conf: 0.942		
24				
25				
26	**SLM-2: Reasoning & Risk Assessment**			
27	**Interpretation**	Likely card feature with operation, but conflicts detected requiring clarification		
28	**Confidence**	low		
29	**Next Step**	proceed		
30	**Risks**			
31				
32				
33	**Scenario Traceability**			
34	**Scenario ID**	**Scenario Name**	**Source (SLM Field)**	**Type**
35	SC-US-CARS-001-001	Successful Activation of Card	slm1.operation.positive	positive
36	SC-US-CARS-001-002	Activation Fails - Card in Invalid State	slm1.operation.negative_state	negative
37	SC-US-CARS-001-003	Activation Fails - Card Not Found	slm1.entity.Card not_found	negative
38	SC-US-CARS-001-004	Activation Blocked by Constraint: Expired Card Blocked	slm1.constraint.expired_card_blocked	business_rule
39	SC-US-CARS-001-005	Activation Blocked by Constraint: Requires Auth	slm1.constraint.requires_auth	business_rule
40	SC-US-CARS-001-006	Activation Blocked by Constraint: Requires Otp	slm1.behaviant.requires_otp	business_rule
41	SC-US-CARS-001-007	Verify beh blocked by Constraint: Zero Balance Required	slm1.constraint.zero_balance_required	business_rule
42	SC-US-CARS-001-008	Verify behavioral effect 'audit log created' is enforced on Cal	slm1.behavior.audit_log_created	validation
43	SC-US-CARS-001-009	Verify behavioral effect 'card status updated' is enforced on	slm1.behavior.card_status_updated	validation
44	SC-US-CARS-001-010	Verify behavioral effect 'email alert card is enforced on Card	slm1.behavior.email_alert_sent	validation
45	SC-US-CARS-001-011	Verify behavioral effect 'sends notification' is enforced on Ca	slm1.behavior.sends_notification	validation
46	SC-US-CARS-001-012	Unauthorized Activation Attempt	security.unauthorized	security
47	SC-US-CARS-001-013	Edge: Concurrent Activation Requests	edge.concurrent	edge
48	SC-US-CARS-001-014	Validate Customer Data During Activation	slm1.entity.Customer.validation	validation

Summary | Scenarios | Test Cases | SLM Trace | +

Fig. 12: SLM-1 requirement extraction output for card activation user story showing extracted feature type, operation, transition, entities, constraints, and behavioural effects with associated confidence scores.

Whereas manual test case development could take several minutes for each of the requirements, the test scenarios and cases developed automatically did not take up much time.

Fig. 12 presents the SLM Trace sheet exported by the pipeline for the Card Activation user story. The upper portion of the sheet records the raw SLM-1 extraction results listing the predicted values for feature type, operation and transition intent alongside their respective confidence scores and the top alternative candidates considered by the classifier. Entity confidence readings for Card (0.802), Customer (0.808), OTP (0.771) and System (0.778) are recorded alongside the detected constraints and their confidence values. The SLM-2 reasoning block beneath it captures the overall interpretation, the assigned confidence level and the next step decision, in this case a proceed outcome despite flagged conflicts, reflecting the rule-based validation logic working as intended. The lower section maps each scenario identifier to its originating SLM field, providing a direct audit trail from requirement attribute to test scenario. This level of traceability is particularly important in banking contexts where test coverage must be demonstrably linked back to specific functional requirements, and the exported sheet makes this linkage available outside the tool itself for review by QA leads or auditors.

5.4 Discussion

Experimental findings prove the successful implementation of the suggested hybrid approach in the process of integrating ML-based requirement interpretation with deterministic scenarios in order to create structured testing artifacts. First of all, it should be

mentioned that the requirement extraction model showed a satisfactory degree of precision but was still not so heavy that it would require the use of large cloud-hosted models; therefore, the use of small transformers is feasible. In terms of the assessment of test case generation quality, it should be stated that the system generates structured test cases which are highly related to the manually developed ones. Although some additional refinement might be necessary on the part of human testers, the generated artifacts save testers a lot of effort in the initial stage of the testing process. Moreover, it should be stated that the suggested approach implies creating test cases with full traceability to requirement attributes. It means that it is easy for testers to trace what test cases were generated on the basis of specific attributes of functional requirements, which is especially useful for the banking domain, as testing artifacts need to cover all aspects of system requirements.[17] In summary, the results indicate that the use of hybrid AI-based systems that involve smaller language models in conjunction with deterministic test logic provides a promising approach to developing automated testing processes. These types of systems are capable of providing substantial benefits in terms of improving the efficiency of the testing process while maintaining necessary transparency.

6. Conclusion

This paper introduced an initially developed framework for test case and scenario generation meant to be used for software testing in banks. The core motivation behind this paper is to tackle two major issues related to AI-assisted software testing: creating a deployable model for extracting requirements and developing a structure for the generated testing output. The suggested framework uses lightweight machine learning algorithms coupled with deterministic algorithms to form a new framework which is able to convert a natural language text of user stories to testing material. The framework includes a series of stages in the testing pipeline, starting from requirement extraction which uses a small language model deployable in on-premises systems. The next stage in the process is validating the extracted requirements in a rule-based quality gate aimed at finding inconsistencies or illogical content. The following stage involves scenario generation using a deterministic algorithm which generates multiple testing scenarios involving positive flow cases, negative cases, constraint validations, and corner cases. The final stage involves generating testing cases which include prerequisites, actions, test data, and desired system behavior. The experiment clearly shows that the described approach is able to provide high-quality extraction of requirement attributes from user stories of a banking software application and create testing artifacts with low effort needed. At the same time, the performance of the requirement extraction module is high in many classification tasks performed. In addition, the pipeline for test creation was able to generate structured test cases that are similar to those created by a tester manually. Yet another significant result of the current research is that traceability was provided for the testing artifacts created. It means that there is a connection established between requirement attributes and scenarios created, as well as steps within test cases. Such traceability enhances visibility within the pipeline for automated testing, allowing testers to understand which testing activities cover which system requirement attributes. Additionally, this research proves the benefits of using hybrid approaches which include machine learning combined with deterministic decision-making processes in generating test cases. On one hand, the former offers possibilities of interpreting natural language requests and processing the meaning of requirements, while the latter helps in achieving deterministic outcomes and ensuring consistent testing artifacts. Combining machine learning and deterministic decision-making provides a chance to create an approach which ensures flexibility and reliability at once – two key factors that make such solutions useful for software testing within enterprise-level software development projects. In conclusion, the results of this research confirm the efficiency of using lightweight models for generating testing artifacts with the help of AI. By using hybrid models and deterministic generation algorithms, it is possible to substantially reduce the amount of work needed for designing initial testing cases without sacrificing any required levels of control over the process. Even though the framework was evaluated using banking software applications, its architecture may also be applicable to other regulated domains such as healthcare, insurance, and government systems where traceability and compliance are important requirements. Future work will involve enlarging the requirement set as well as refining the scenario-generation mechanism in order to handle complex business scenarios involving several steps. Other experiments that will be carried out include conducting tests using large data sets and different banks in order to evaluate the scalability and generalization capability of the proposed framework. In summary, this study provides insights for building an AI-supported system that can aid software developers working in the banking industry.

CRediT Author Contribution Statement

Rinku Ashish Badgujar: Conceptualization, Data Curation, Formal analysis; Investigation; Methodology; Validation; Visualization; Writing – Original draft; Writing – Review & editing. **Prerna Pareek:** Data curation; Formal analysis; Investigation; Project administration; Resources; Software; Supervision;

Validation; Writing – Review & editing.

Funding Declaration
This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Data Availability Statement
No data were generated or analysed during the current study. Therefore, data sharing is not applicable to this article.

Conflict of Interest
There is no conflict of interest.

Artificial Intelligence (AI) Use Disclosure
The authors declare that no artificial intelligence (AI) or AI-assisted technologies were used in the writing, editing, or preparation of this manuscript, nor were any images generated using AI tools.

Supporting Information
Not applicable

References

[1] M. F. Granda, O. Parra, M. F. Alba-Sarango, Towards a Model-Driven Testing Framework for GUI Test Cases Generation from User Stories, 16th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2021), 2021, 453–460, doi: 10.5220/0010499004530460.

[2] H. Hourani, A. Hammad, M. Lafi, The Impact of Artificial Intelligence on Software Testing, 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT), 2019, 565–570, doi: 10.1109/JEEIT.2019.8717439.

[3] Z. Khaliq, S. U. Farooq, D. A. Khan, Artificial Intelligence in Software Testing: Impact, Problems, Challenges and Prospect, *arXiv preprint*, 2022, doi: 10.48550/arXiv.2201.05371.

[4] A. Ramadan, H. Yasin, B. Pektas, The Role of Artificial Intelligence and Machine Learning in Software Testing, *arXiv preprint*, 2024, arXiv:2409.02693, doi: 10.48550/arXiv.2409.02693.

[5] H. V. Pandhare, From Test Case Design to Test Data Generation: How AI Is Redefining QA Processes, *International Journal of Engineering and Computer Science*, 2024, **13**, doi: 10.18535/ijecs.v13i12.4956.

[6] M. Islam, F. Khan, S. Alam, M. Hasan, Artificial Intelligence in Software Testing: A Systematic Review, *2023* IEEE Region 10 Conference (TENCON), 2023, 524–529, doi: 10.1109/TENCON58879.2023.10322349.

[7] H. Ayenew, M. Wagaw, Software Test Case Generation Using Natural Language Processing (NLP): A Systematic Literature Review, *Artificial Intelligence Evolution*, 2024, 1–10, doi: 10.37256/aie.5120243220.

[8] I. K. Raharjana, D. Siahaan, C. Fatichah, User Stories and Natural Language Processing: A Systematic Literature Review, *IEEE Access*, 2021, **9**, 53811–53826, doi: 10.1109/ACCESS.2021.3070606.

[9] H. Mollah, P. van den Bos, From User Stories to End-to-End Web Testing, *2023* IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2023, 140–148, doi: 10.1109/ICSTW58534.2023.00036.

[10] C. Wang, F. Pastore, A. Goknil, L. C. Briand, Automatic Generation of Acceptance Test Cases from Use Case Specifications: An NLP-Based Approach, *IEEE Transactions on Software Engineering*, 2022, **48**, 585–616, doi: 10.1109/TSE.2020.2998503.

[11] M. Baqar, R. Khanda, The Future of Software Testing: AI-Powered Test Case Generation and Validation, *Intelligent Computing*, 2025,**1424**, doi: 10.1007/978-3-031-92605-1_18.

[12] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, N. Sundaresan, Unit Test Case Generation with Transformers and Focal Context, *arXiv preprint*, 2020, doi: 10.48550/arXiv.2009.05617.

[13] W. C. Ouédraogo, A. K. Kaboré, H. Tian, Y. Song, A. Koyuncu, J. Klein, D. Lo, T. F. Bissyandé, Large-Scale, Independent and Comprehensive Study of the Power of LLMs for Test Case Generation, *arXiv preprint*, 2024, doi: 10.48550/arXiv.2407.00225.

[14] F. S. Manzoni, R. Rodrigues, A. C. O. Rocha, Exploring the Use of ChatGPT for the Generation of User Story Based Test Cases: An Experimental Study, Proceedings of the 27th International Conference on Enterprise Information Systems (ICEIS 2025), 2025, doi: 10.5220/0013398700003929.

[15] M. Brunetto, G. Denaro, L. Mariani, M. Pezzè, On Introducing Automatic Test Case Generation in Practice: A Success Story and Lessons Learned, *Journal of Systems and Software*, 2021, **176**, 110933, doi:10.1016/j.jss.2021.110933.

[16] J. G. Santos, R. S. P. Maciel, AutomTest 3.0: An Automated Test-Case Generation Tool from User Story Processing Powered with LLMs, *Simpósio Brasileiro de Engenharia de Software (SBES)*, 2024, 769–774, doi: 10.5753/sbes.2024.3603.

[17] C. Tao, J. Gao, T. Wang, Testing and Quality Validation for AI Software—Perspectives, Issues, and Practices, *IEEE Access*, 2019, **7**, 120164–

120175, doi: 10.1109/ACCESS.2019.2937107.
[18] M. Olsthoorn, More Effective Test Case Generation with Multiple Tribes of AI, Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, 2022, 286–290, doi: 10.1145/3510454.3517066.
[19] S. Alagarsamy, C. Tantithamthavorn, A. Aleti, A3Test: Assertion-Augmented Automated Test Case Generation, *Information and Software Technology*, 2024, **176**, 107565, doi: 10.1016/j.infsof.2024.107565.
[20] S. Qazi, M. Zafar, N. Aslam, M. Khan, Software Quality Assurance Using Artificial Intelligence Techniques: A Survey of the Software Industry of Pakistan, *Journal of Hunan University Natural Sciences*, 2023, **50**, 89–106, doi: 10.55463/issn.1674-2974.50.12.1.
[21] M. Bagherzadeh, N. Kahani, L. Briand, Reinforcement Learning for Test Case Prioritization, *IEEE Transactions on Software Engineering*, 2022, **48**, 2836–2856, doi: 10.1109/TSE.2021.3070549.
[22] A. Agarwal, P. Lohia, S. Nagar, K. Dey, D. Saha, Automated Test Generation to Detect Individual Discrimination in AI Models, *arXiv preprint*, 2018, arXiv:1809.03260, doi: 10.48550/arXiv.1809.03260.
[23] A. Arcuri, RESTful API Automated Test Case Generation, 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2017, 9–20, doi: 10.1109/QRS.2017.11.
[24] Y. Bajaj, M. K. Samal, Accelerating Software Quality: Unleashing the Power of Generative AI for Automated Test-Case Generation and Bug Identification, *International Journal for Research in Applied Science and Engineering Technology*, 2023, **11**, 345–350, doi: 10.22214/ijraset.2023.54628.
[25] A. Bandi, H. S. T. Nukala, B. Tatavarthi, A. Boggavarapu, Automated Test Case Generation for Software Testing Using Generative AI, *International Conference on Computers and Their Applications*, 2025, **2435**, doi: 10.1007/978-3-031-92178-0_7.
[26] J. Wang, B. Suleiman, M. J. Alibasa, Automated Unit Test Case Generation: A Systematic Literature Review, *CoRR*, *arXiv preprint arXiv:2504.20357*, 2025, doi: 10.48550/arXiv.2504.20357.
[27] S. Bhatia, T. Gandhi, D. Kumar, P. Jalote, Unit Test Generation Using Generative AI: A Comparative Performance Analysis of Autogeneration Tools, Proceedings of the 1st International Workshop on Large Language Models for Code, 2024, 54–61, doi: 10.1145/3643795.3648396.
[28] M. Lafi, T. Alrawashed, A. M. Hammad, Automated Test Cases Generation from Requirements Specification, 2021 International Conference on Information Technology (ICIT), 2021, 852–857, doi: 10.1109/ICIT52682.2021.9491761.

Publisher Note: The views, statements, and data in all publications solely belong to the authors and contributors. GR Scholastic is not responsible for any injury resulting from the ideas, methods, or products mentioned. GR Scholastic remains neutral regarding jurisdictional claims in published maps and institutional affiliations.