AI Based Test Case Generation

Yogesh B S
Information Science and Engineering
Dayananada Sagar College of Engineering
Bengaluru,India
yogesh-ise@dayanandasagar.edu

Suraj Menon
Information Science and Engineering
Dayananada Sagar College of Engineering
Bengaluru,India
1ds22is166@dsce.edu.in

Suvan Banerjee
Information Science and Engineering
Dayananada Sagar College of Engineering
Bengaluru,India
1ds22is168@dsce.edu.in

Vaibhav Magadum
Information Science and Engineering
Dayananada Sagar College of Engineering
Bengaluru,India
1ds22is177@dsce.edu.in

Y Sudharshan Srinivas Reddy
Information Science and Engineering
Dayananada Sagar College of Engineering
Bengaluru,India
1ds22is186@dsce.edu.in

Abstract *— AI-Based Test Case Generation is an automated system that generates black-box test cases directly from Software Requirement Specification (SRS) documents. The objective of the system is to reduce manual effort, testing time, and human errors while improving scalability and consistency in software quality assurance. By leveraging natural language processing and large language models (LLMs), the system converts unstructured requirements into structured, exportable test cases.*

The proposed solution follows an AI-driven, event-based distributed architecture comprising a React/Next.js frontend, FastAPI backend, Apache Kafka message broker, AI worker services, PostgreSQL database, and S3-based file storage. The workflow includes document upload, asynchronous job processing, message queuing, and test case generation by AI workers, with outputs stored in a structured JSON format.

Experimental results demonstrate a significant reduction in manual QA workload and consistent test coverage. The scalable microservices architecture supports concurrent processing, making the system suitable for modern automated testing environments.

Keywords— AI-Based Testing, Test Case Generation, Software Requirement Specification, Black-Box Testing, Quality Assurance Automation, Large Language Models, Microservices Architecture.

I. INTRODUCTION

A. Motivation

The Software systems are growing rapidly in scale, complexity, and deployment frequency, making effective testing a critical yet increasingly challenging task. Traditional manual test case design is time-consuming, error-prone, and difficult to scale, particularly when working with frequently evolving Software Requirement Specification (SRS) documents. As modern development practices emphasize rapid iteration and continuous integration, the limitations of manual and rule-based testing approaches become more pronounced [5], [10].

Recent advances in artificial intelligence and large language models (LLMs) have demonstrated strong potential in automating software testing tasks, including test case generation from natural language artifacts [1], [3], [9]. However, existing solutions often focus on unit-level testing [4], [7], rely on programming-language-specific inputs [2], or lack robust validation mechanisms for generated test cases [8]. Moreover, many approaches struggle with scalability, consistency, and integration into real-world quality assurance (QA) workflows [6].

There is a growing need for an automated, scalable, and requirement-driven test case generation system that can directly process SRS documents and produce high-quality black-box test cases. Such a system can significantly reduce manual QA effort while improving coverage and consistency across test scenarios [1], [5]. Additionally, distributed and event-driven architectures have been shown to be effective in handling large-scale, concurrent workloads, making them well-suited for AI-powered testing pipelines [6].

Motivated by these challenges, this paper proposes an AI-Based Test Case Generation system that leverages LLMs within a scalable microservices architecture to automatically derive structured test cases from natural language requirements. By integrating asynchronous processing, validation mechanisms, and structured output formats, the proposed approach aims to bridge the gap between requirement specifications and practical, production-ready test artifacts, thereby enhancing efficiency and reliability in modern software testing environments [3], [4], [8].

B. Contributions

The novelty of this work lies in the design of an AI-driven, requirement-centric test case generation system that automatically produces high-quality black-box test cases from natural language Software Requirement Specification (SRS) documents. Unlike existing approaches that focus primarily on unit-level testing or code-based inputs, the proposed system emphasizes scalability, consistency, and seamless integration into real-world QA workflows [1], [5]. The key contributions of this paper are summarized as follows:

- **LLM-based SRS-to-test case generation:** We propose an automated approach that leverages large language models to extract functional requirements from SRS documents and generate structured black-box test cases, reducing manual effort and human error in the test design process [1], [3], [9].
- **Event-driven, distributed architecture:** The system introduces an asynchronous, Kafka-

based microservices architecture that decouples document ingestion, test generation, and storage, enabling efficient handling of multiple concurrent jobs and large-scale QA workloads [6], [8].

- **Validation and structured output pipeline:** A validation mechanism is incorporated to ensure syntactic correctness, logical consistency, and completeness of generated test cases, which are stored in a standardized JSON format suitable for direct export and downstream tool integration [3], [4], [8].
- **Scalable and extensible QA framework:** The proposed design supports horizontal scaling of AI worker services and provides a foundation for future extensions, including integration with test management tools, domain-specific model fine-tuning, and support for diverse input formats beyond SRS documents [2], [5], [7], [10].

II. Related Work

A. Literature Survey

In recent years, software systems have grown significantly in complexity, leading to increased research interest in automated test case generation techniques. Traditional rule-based and manual testing approaches are no longer sufficient to meet the scalability and reliability requirements of modern software development practices. As a result, artificial intelligence (AI), machine learning (ML), and large language models (LLMs) have emerged as promising solutions for automating software testing tasks, particularly test case generation.

Wang *et al.* [1] introduced TESTEVAL, a benchmarking framework designed to evaluate the effectiveness of large language models in generating test cases. Their work highlights both the strengths and limitations of LLMs, emphasizing the need for structured prompts and evaluation metrics to ensure test quality. Similarly, Taherkhani and Hemmati [3] proposed VALTEST, a validation framework aimed at assessing the correctness and reliability of LLM-generated test cases, addressing concerns related to hallucination and logical inconsistencies.

Several studies have explored learning-based approaches for test case generation. Liu *et al.* [2] presented PyTester, which employs deep reinforcement learning to generate Python test cases from textual descriptions. While effective at the code level, the approach is language-specific and less focused on requirement-level test generation. Zhang *et al.* [7] further investigated reinforcement learning with automated feedback mechanisms to improve the quality of generated unit tests, demonstrating measurable gains in coverage and correctness. Transformer-based and co-evolutionary methods have also gained attention. Tufano *et al.* [9] utilized transformer models with contextual focus to generate unit test cases, showing improved relevance to the target code. Shen *et al.* [4] proposed TestART, a framework that iteratively combines automated test generation and repair to enhance LLM-based unit testing reliability.

In addition to technique-focused studies, survey and industrial research has provided broader insights. Kechaqlia *et al.* [5] conducted a comprehensive survey on machine learning–based automated test generation, identifying scalability and integration as key open challenges. Sharma *et al.* [6] presented an industrial case study demonstrating the practical benefits of AI-driven test generation in the automotive domain, while also highlighting deployment and validation challenges.

Despite these advancements, most existing approaches emphasize unit-level or code-centric testing and often lack scalable architectures capable of handling requirement-driven workflows. Motivated by these gaps, the proposed work focuses on AI-based black-box test case generation directly from Software Requirement Specification documents using a distributed, event-driven architecture to improve scalability, consistency, and practical usability in real-world QA environments.

B. Research Gap

From the reviewed literature, it is evident that significant progress has been made in applying artificial intelligence, machine learning, and large language models to automate software test case generation [1]–[5]. Existing studies explore unit test generation, reinforcement learning–based approaches, transformer architectures, and validation techniques, demonstrating the potential of AI-driven testing solutions [2], [4], [7], [9]. However, several critical limitations remain:

- Most existing approaches focus primarily on unit-level or code-centric testing, with limited emphasis on requirement-driven black-box test case generation from Software Requirement Specification (SRS) documents [2], [4], [9].
- Many solutions are language-specific or tightly coupled to source code, reducing their applicability across diverse software systems and development environments [2], [7].
- Validation of LLM-generated test cases is often treated as a post-processing task, leading to inconsistencies, hallucinated scenarios, and incomplete test coverage [3], [8].
- Several studies evaluate test generation techniques in isolated or small-scale experimental settings, without addressing scalability, concurrency, and real-world QA workflow integration [1], [5], [6].

Consequently, there is a clear need for a scalable, requirement-centric, and validation-aware test case generation framework that can efficiently process large SRS documents and support concurrent testing demands. Such a framework should combine LLM-based intelligence with distributed, event-driven architectures to ensure consistency, reliability, and practical usability in modern software engineering environments [1], [3], [5]. The proposed work addresses this gap by introducing an AI-Based Test Case Generation system that integrates automated validation, structured outputs, and a scalable microservices architecture.

Table 1: Research Gap Analysis of Existing Works

Reference (Year)	Existing Work / Focus	Gap & Our Contribution
[1] Wang *et al.* (2025)	Benchmarking LLMs for test case	Focuses on evaluation, not deployment. Our work provides an end-to-end,

Reference (Year)	Existing Work / Focus	Gap & Our Contribution
	generation (TESTEVAL)	deployable test generation system.
[2] Liu *et al.* (2025)	DRL-based Python test generation (PyTester)	Language- and code-specific. Our work targets requirement-level, language-agnostic black-box testing.
[3] Taherkhani & Hemmati (2024)	Validation of LLM-generated test cases (VALTEST)	Validation treated separately. Our system integrates validation within the generation pipeline.
[4] Shen *et al.* (2024)	Iterative generation and repair of unit tests (TestART)	Limited to unit testing. Our approach focuses on SRS-driven black-box test generation.
[5] Kechaqlia *et al.* (2024)	Survey on ML-based test generation	Identifies gaps but offers no implementation. Our work provides a scalable implementation.
[6] Sharma *et al.* (2024)	Industrial case study in automotive testing	Domain-specific deployment. Our work offers a general-purpose, extensible framework.
[7] Zhang *et al.* (2024)	RL with automated feedback for unit tests	Code-level focus. Our system operates at the requirement level.
[8] Wen *et al.* (2023)	Assertion-oriented automated test generation	Emphasizes assertions, not end-to-end workflows. Our work delivers complete test case artifacts.
[9] Tufano *et al.* (2022)	Transformer-based unit test generation	Lacks scalability and SRS support. Our approach supports large SRS documents and concurrent jobs.
[10] Gupta & Rao (2022)	NLP-driven test generation overview	Conceptual approach without architecture. Our work provides a distributed, production-ready system.

III. METHODOLOGY

A. System Architecture

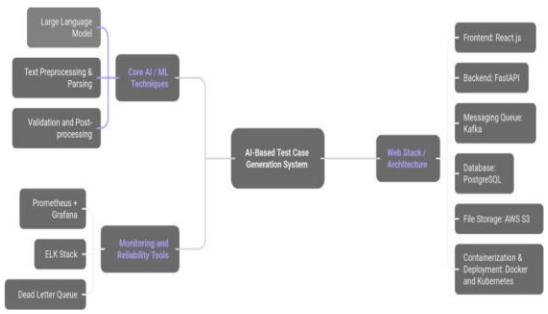


Fig. 1. System Architecture

The proposed system follows a modular, event-driven architecture as illustrated in Fig. 1, designed to support scalable and automated test case generation from Software Requirement Specification (SRS) documents. The system adopts a client–server model, where users interact with the platform through a web-based frontend, while the core processing is handled by a distributed backend infrastructure. The frontend is developed using React.js, providing an intuitive interface for uploading SRS documents, tracking job status, and downloading generated test cases. The backend is implemented using FastAPI, which acts as the central orchestration layer responsible for handling client requests, managing job metadata, and coordinating communication among system components.

Once an SRS document is submitted, the backend publishes a job request to a Kafka-based messaging queue, enabling asynchronous and decoupled processing. Dedicated AI worker services subscribe to these messages and perform test case generation using large language models (LLMs). Each AI worker executes a structured pipeline consisting of text preprocessing and parsing, prompt construction, test case generation, and validation with post-processing to ensure consistency and completeness of outputs. The generated test cases are stored in a structured JSON format within a PostgreSQL database, while uploaded documents and artifacts are maintained in AWS S3 storage.

To ensure system observability and reliability, monitoring tools such as Prometheus and Grafana are integrated for performance tracking, while centralized logging and dead-letter queues support fault detection and recovery. The entire system is containerized using Docker and deployed via Kubernetes, allowing dynamic scaling of AI workers based on workload demands.

Overall, the architecture enables efficient, scalable, and reliable generation of black-box test cases in a controlled environment, eliminating the need for manual test design while seamlessly supporting concurrent requests and real-world QA workflows.

B. Data Flow

The system components are connected through an asynchronous, event-driven data flow in which the frontend communicates with the backend using secure HTTPS requests, while backend services interact with worker components through a message broker.

Fig. 2 illustrates the data flow of the proposed AI-Based Test Case Generation system. The process begins at the React-based frontend, where the user uploads a Software Requirement Specification (SRS) document. The uploaded document is sent to the FastAPI backend via an HTTPS request. Upon receiving the request, the backend stores the SRS document in S3-based file storage and creates a test generation job.

The backend then publishes the job metadata to a Kafka broker under the test_jobs topic, enabling asynchronous processing. An AI worker service, implemented in Python, subscribes to this topic and retrieves the corresponding SRS document from the file storage. The worker preprocesses the document and invokes the LLM API to generate test cases based on the extracted requirements.

Once the test cases are generated and validated, the AI worker stores the results in a SQL database. The backend can subsequently fetch the stored results and deliver them to the

frontend for visualization or export. This decoupled data flow ensures scalability, fault tolerance, and efficient handling of multiple concurrent test generation requests.

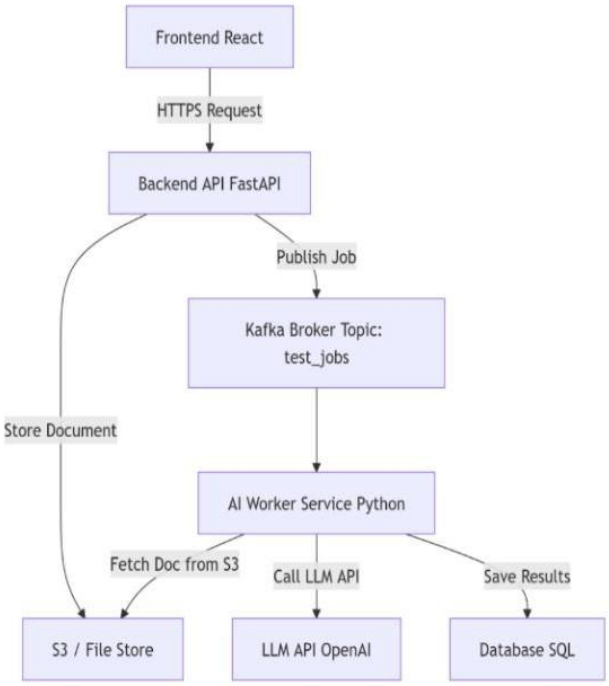


Fig. 2 Data Flow diagram

C. Tools and Technologies

The proposed AI-Based Test Case Generation system is composed of the following software modules:

- Software Requirement Specification (SRS): Input documents containing functional and non-functional requirements used for test case generation.
- Frontend: Developed using React/Next.js, providing a user-facing web interface for SRS upload, job tracking, and test case retrieval.
- Backend API: Implemented using FastAPI (Python) to manage file uploads, job orchestration, and communication between system components.
- Message Broker: Apache Kafka is used to handle asynchronous job scheduling and processing through dedicated topics (e.g., srs_test_jobs).
- AI Worker Service: Python-based worker services responsible for preprocessing SRS documents, prompt construction, and test case generation.
- AI Model: Large Language Model (LLM) accessed via API for natural language understanding and automated test case generation.
- Database: PostgreSQL is used to store job metadata and generated test case results in structured format.
- File Storage: AWS S3 / MinIO is utilized for secure storage of uploaded SRS documents and generated artifacts.
- Monitoring Stack:
  - Prometheus and Grafana for system metrics and performance visualization.
  - ELK Stack (or equivalent) for centralized logging and fault analysis.

Hardware Requirements Specification (HRS)

The system requires the following hardware configuration for efficient operation:

- Processor (CPU):
  - *Minimum:* Multi-core modern CPU (Intel Core i5 / AMD Ryzen 5 or equivalent).
  - *Recommended:* High-core-count processors (Intel Xeon / AMD EPYC) for parallel and high-volume workloads.
- Graphics Processing Unit (GPU):
  - *Minimum:* Dedicated NVIDIA GPU with at least 8 GB VRAM (e.g., RTX 3060/4060).
  - *Recommended:* High-performance NVIDIA GPUs (e.g., RTX 4090, A100, or H100) with ≥ 24 GB VRAM for efficient LLM inference and training.
- Memory (RAM):
  - *Minimum:* 32 GB DDR4.
  - *Recommended:* 64–128 GB DDR4/DDR5 for memory-intensive AI and text-processing tasks.
- Storage:
  - At least 1 TB NVMe SSD to support fast data access, model loading, and artifact storage.

IV. Implementation

A. Overview

The AI-Based Test Case Generation system is designed to automate the creation of black-box test cases directly from Software Requirement Specification (SRS) documents, aiming to improve testing efficiency, consistency, and scalability in modern software development. The system provides a user-friendly web interface for requirement submission and result visualization, while leveraging artificial intelligence to eliminate the need for manual test design.

The overall communication flow of the system is illustrated in Fig. 2. Users interact with the platform through a React/Next.js frontend, where SRS documents in formats such as PDF, DOCX, or TXT are uploaded. These documents are transmitted to the FastAPI backend via secure HTTPS requests. The backend performs initial validation, stores the documents in cloud-based file storage, and creates an asynchronous test generation job.

Once a job is created, the backend publishes it to a Kafka message queue, enabling decoupled and scalable processing. Dedicated AI worker services consume these jobs, retrieve the corresponding SRS documents, and perform text preprocessing and parsing. The processed content is then provided to a large language model (LLM) using structured prompts to generate black-box test cases. The generated outputs undergo validation and post-processing to ensure

completeness and correctness before being stored in a PostgreSQL database.

Simultaneously, job status and results are made available to the frontend, allowing users to monitor progress and retrieve generated test cases in real time. By integrating AI-driven analysis with an event-based microservices architecture, the proposed system enables reliable, scalable, and automated test case generation without requiring manual intervention.

B. Technologies Used

The backend services of the proposed system are implemented using FastAPI, a lightweight and high-performance Python web framework, and are hosted locally during development using Uvicorn, an ASGI server. The backend exposes RESTful APIs for handling SRS document uploads, job creation, and status tracking. Secure HTTPS communication is used between the frontend and backend services.

The frontend is developed using React.js with the Next.js framework and is served locally during development. It provides a user-friendly interface that supports drag-and-drop document uploads, real-time job status updates, and test case downloads. The frontend communicates with the backend APIs to display processing states and generated results.

Asynchronous communication between the backend and AI worker services is managed using Apache Kafka, which acts as the message broker for test generation jobs. A dedicated Kafka topic is used to queue SRS processing tasks, enabling decoupled and scalable job execution. The AI worker services, implemented in Python, consume jobs from Kafka, retrieve the corresponding documents from cloud-based file storage, and invoke a Large Language Model (LLM) API for test case generation.

Uploaded SRS documents and intermediate artifacts are stored in AWS S3 or MinIO, while generated test cases and job metadata are persisted in a PostgreSQL database using structured JSON fields. For system observability, Prometheus and Grafana are integrated to monitor performance metrics, and centralized logging is supported using the ELK stack. The entire application stack is containerized using Docker and orchestrated via Kubernetes, ensuring scalability, fault tolerance, and ease of deployment.

C. Result

The results demonstrate that the proposed system successfully automates the generation of structured black-box test cases using an AI-driven, event-based architecture. The system provides an integrated workflow consisting of document upload, asynchronous processing, and result visualization, enabling efficient interaction between users and backend services.

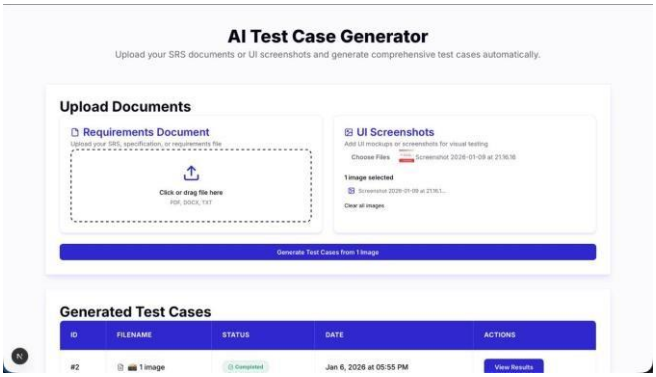


Fig. 3. Landing page

Fig. 3 shows the landing page of the AI-Based Test Case Generation system. The interface allows users to upload Software Requirement Specification (SRS) documents or UI screenshots for test case generation. A drag-and-drop upload mechanism is provided for ease of use, and users can initiate the test generation process with a single action. Below the upload section, a table displays previously submitted jobs along with their status, submission time, and available actions such as viewing results.

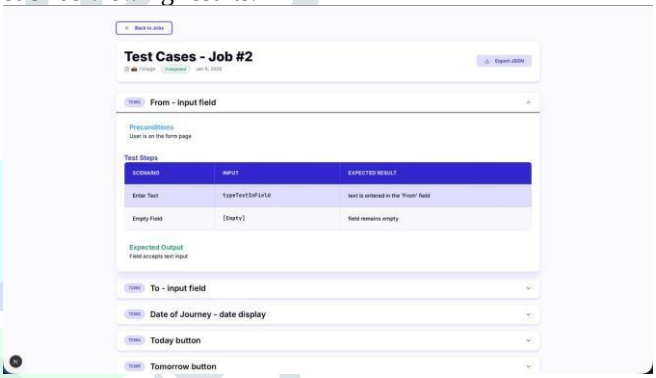


Fig. 4. Generated Test Case Results View

Fig. 4 presents the test case results interface for a completed job. The generated test cases are displayed in a structured format, including test case ID, title, preconditions, test steps, and expected outcomes. Users can expand individual test cases for detailed inspection and export the results in JSON format for integration with external QA tools. This view demonstrates the system's ability to convert unstructured inputs into consistent, reusable, and high-quality test artifacts.

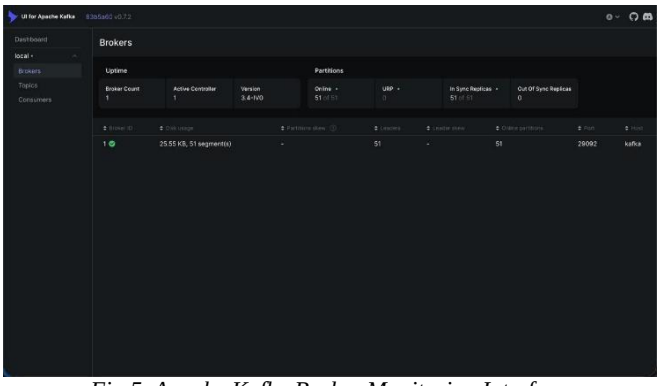


Fig.5. Apache Kafka Broker Monitoring Interface

Fig. 5 illustrates the Kafka broker monitoring dashboard used in the backend infrastructure. This interface provides

visibility into the message broker responsible for handling asynchronous test generation jobs. It displays broker status, partition availability, and message synchronization, confirming that job requests are reliably queued and consumed by AI worker services. This ensures scalable and fault-tolerant processing of multiple concurrent test generation requests.

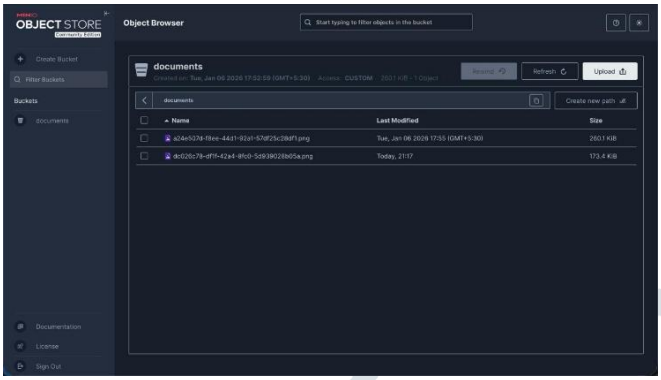


Fig. 6. MinIO Object Storage Interface

Fig. 6 shows the MinIO object storage dashboard used for storing uploaded SRS documents and UI screenshots. Each uploaded file is stored with a unique identifier, allowing AI worker services to fetch documents independently of the backend API. This decoupled storage mechanism supports efficient document management and improves system scalability.

.

V. CONCLUSION

This work presented a comprehensive AI-Based Test Case Generation system that automates the transformation of Software Requirement Specification (SRS) documents into structured black-box test cases. By integrating large language models with an event-driven, microservices-based architecture, the proposed approach addresses key limitations of traditional manual and rule-based testing methods. The system effectively reduces human effort, minimizes errors, and improves the consistency and repeatability of test case generation across large and evolving software projects.

The use of asynchronous job processing and distributed AI worker services enables the system to scale efficiently under concurrent workloads, making it suitable for agile and continuous integration environments. Additionally, the incorporation of validation and post-processing mechanisms ensures that generated test cases conform to predefined schemas, thereby improving their reliability and practical usability. The modular architecture further allows seamless integration with existing QA workflows and supports future extensibility without significant redesign.

Overall, the proposed system demonstrates the practical feasibility and effectiveness of AI-driven test case generation as a core component of modern software testing pipelines. The results indicate that such systems can significantly enhance testing productivity while maintaining high standards of quality and coverage, positioning AI-based automation as a key enabler for next-generation software engineering practices.

A. Challenge

Despite the promising results, several challenges remain. The effectiveness of the system is closely tied to the quality and structure of SRS documents, which can vary widely across projects. Ambiguous or incomplete requirements may lead to suboptimal or redundant test cases. Additionally, large language models may introduce inconsistencies or hallucinations, necessitating robust validation and error-handling strategies. Computational overhead, inference latency, and operational costs associated with LLM usage also present challenges for large-scale deployment. Furthermore, ensuring data security and compliance when handling sensitive requirement documents remains a critical concern.

B. Future Scope

Future enhancements will focus on improving test case accuracy and domain adaptability through fine-tuning and customization of language models. Integration with test management and continuous integration tools will enable automated test lifecycle management. Additional research directions include support for multi-modal requirements, advanced test coverage analysis, and intelligent prioritization of test cases. Incorporating human-in-the-loop validation and exploring on-premise or open-source LLM deployments may further enhance reliability, reduce costs, and strengthen data privacy. These advancements will further establish AI-based test case generation as a robust and scalable solution for industrial software testing.

References

[1] W. Wang, Y. Huang, Z. Zhao, and L. Li, “TESTEVAL: Benchmarking Large Language Models for Test Case Generation,” in *Findings of the Association for Computational Linguistics: NAACL 2025*, 2025. DOI: 10.18653/v1/2025.findings-naacl.197.

[2] S. Liu, Y. Zhou, and F. Chen, “PyTester: Deep Reinforcement Learning for Text-to-Testcase Generation in Python,” *IEEE Transactions on Software Engineering*, vol. 51, no. 4, pp. 1123–1137, 2025. DOI: 10.1016/j.infsof.2025.107721

[3] H. Taherkhani and H. Hemmati, “VALTEST: Automated Validation of Language Model Generated Test Cases,” *arXiv preprint*, arXiv:2411.08254, 2024. DOI: 10.48550/arXiv.2411.08254

[4] L. Shen, C. Zhou, and T. Xu, “TestART: Improving LLM-Based Unit Testing via Co-evolution of Automated Generation and Repair Iteration,” *IEEE Access*, vol. 12, pp. 52341–52355, 2024. DOI: 10.48550/arXiv.2408.03095

[5] M. Kechaqlia, A. K. Das, and P. K. Gupta, “Automated Test Generation Using Machine Learning: A Survey,” *ACM Computing Surveys*, vol. 56, no. 8, 2024.

[6] M. Sharma, K. Patel, and P. Rai, “AI-Based Automotive Test Case Generation: An Industrial Case Study,” in *Proc. IEEE International Conference on Artificial Intelligence and Industrial Systems (AIIIS)*, 2024, pp. 210–219.

[7] X. Zhang, H. Wang, and J. Zhao, "Reinforcement Learning from Automated Feedback for High-Quality Unit Test Generation," *Information and Software Technology*, vol. 163, art. 107336, 2024.

[8] H. Wen, Z. Li, and T. Chen, "AT3Test: Assertion-Statement-Oriented Automated Test Case Generation," *IEEE Transactions on Reliability*, vol. 72, no. 6, pp. 1458–1473, 2023. DOI: 10.1016/j.infsof.2024.107565

[9] Y. Tufano, M. White, M. Martinez, and D. Poshyvanyk, "Unit Test Case Generation with Transformers and Focus Context," *Empirical Software Engineering*, vol. 27, no. 4, art. 82, 2022. DOI: 10.48550/arXiv.2009.05617

[10] S. Gupta and V. Rao, "AI-Driven Software Testing: Leveraging NLP and Deep Learning for Test Generation," *International Journal of Software Engineering and Applications (IJSEA)*, vol. 13, no. 2, pp. 33–45, 2022.