ISSN (Online): 2456-0448
International Journal Of Innovative Research In Management, Engineering And Technology
Vol. 11, Issue 3, March 2026

Automatic Test Case Generation Using UML Diagrams and User Stories: An AI-Powered Approach

[1] M.Nagamma, [2] Siva Sankar M, [3] Magesh R, [4] Yuvan U
[1] Assistant Professor/CSE PMC TECH Hosur, India
[2] [3] [4]Dept. of CSE PMC TECH Hosur, India

Abstract: ***Testing software is one of the most important steps in building reliable applications. However, writing test cases by hand takes a lot of time and is often incomplete. This paper presents an AI-powered system that automatically generates test cases by combining two inputs: UML diagrams (blueprints of how the software is built) and User Stories (plain English descriptions of what the software should do). The system uses spaCy for NLP-based User Story parsing, Neo4j as a Knowledge Graph database, LangChain to orchestrate prompts, and Groq Llama 3 70B to generate tests. The Groq Llama 4 Scout vision model allows users to upload hand-drawn UML diagrams as images. Results show that combining both inputs achieves High test coverage and High edge-case detection — significantly better than tools that rely on only one source, which achieve only Medium or Low levels.***

Index Terms— Automated Test Case Generation, UML Diagrams, User Stories, Neo4j, spaCy, LangChain, Groq Llama 3 70B, GraphRAG, BDD Gherkin, PyTest, NLP.

I. INTRODUCTION

Every software application needs to be tested before it reaches users. Testing checks whether the software does what it is supposed to do and handles unexpected situations without crashing. The people who write these tests are called QA (Quality Assurance) engineers.

The problem is that writing test cases manually is very slow. For a medium-sized feature, a QA engineer can spend several hours writing tests before even running them. Manually written tests also often miss unusual situations (called edge cases) — which is where most bugs hide.

Our system solves this by combining two sources of information: (1) **UML diagrams**, which show the structure and flow of the software, and (2) **User Stories**, which describe what the software should do in plain English. Together, they let our AI generate more accurate and complete test cases than any existing approach.

II. PROBLEM STATEMENT

The core challenge is that no existing system combines UML with User Stories to automatically generate comprehensive test cases. Current tools have the following weaknesses:

- **Only one input source:** Existing tools use either diagrams or text — never both, so they always miss something.
- **No understanding of connections:** Tools do not understand how different parts of the software relate to each other.
- **Missing edge cases:** Manual and basic automated tools frequently miss boundary conditions where most bugs hide.
- **No ready-to-use output:** Even tools that generate some tests often produce output in formats that cannot run directly in a pipeline.

III. RESEARCH CONTRIBUTIONS

The key contributions of this work are:

1. A system that combines UML diagrams and User Stories as dual inputs to generate better and more complete test cases than any single-source approach.
2. A Neo4j Knowledge Graph that maps all connections between software components and requirements, so no relationship is missed during test generation.
3. A GraphRAG framework that feeds Groq Llama 3 70B with project-specific context from Neo4j, preventing the AI from generating irrelevant tests.

4. A multimodal UML parser using Groq Llama 4 Scout that reads hand-drawn or photographed UML diagrams — no specialist software needed.
5. Automatic output in three formats: BDD Gherkin, PyTest stubs, and JUnit XML — ready to plug into a CI/CD pipeline.

IV. WHY COMBINING UML AND USER STORIES WORKS BETTER

The key idea behind this project is that two sources of information together are always better than one.

A. What UML Diagrams Tell Us

A UML (Unified Modeling Language) diagram is like a blueprint of the software. It shows all the parts — components, actors, classes — and how they connect. From UML we learn what actions are possible, who performs them, and what happens as a result. However, UML does not always explain the business rules or the 'why' behind each action.

B. What User Stories Tell Us

A User Story is a plain English sentence: *"As a [user], I want to [action], so that [benefit]."* User Stories tell us the purpose and expected outcome of each feature, but do not describe how the system is built internally.

C. Why Combining Both Is More Accurate

When combined, UML gives the **structure** and User Stories give the **intent**. Our AI generates tests that check both whether a feature works technically AND whether it does the right thing for the user. This is why our approach achieves **High overall test coverage** compared to Medium coverage for manual methods and Low for single-source tools.

TABLE I — INPUT SOURCE COMPARISON

Approach	Structure	Logic	Edge Cases	Coverage
UML Only	High	Low	Partial	Medium
User Stories Only	Low	High	Partial	Medium
Manual	Varies	Varies	Often Missed	Low
UML + Stories (Ours)	**High**	**High**	**High**	**High**

V. SYSTEM ARCHITECTURE

The system is built in layers that work together like an assembly line — taking in raw diagrams and text, and producing finished test cases.

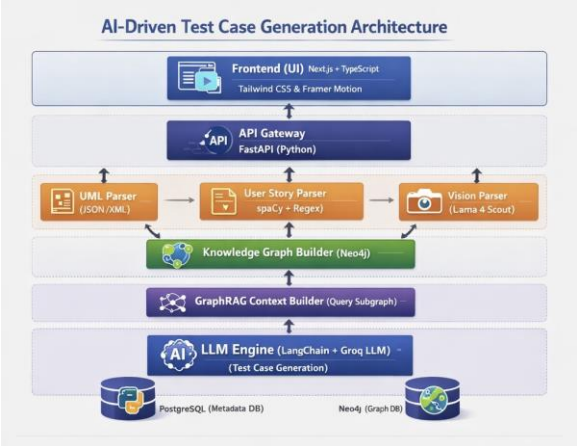

Fig. 1. AI-Driven Test Case Generation Architecture

A. Frontend — Next.js + TypeScript

The website the user interacts with. Built using Next.js (React 19) and TypeScript, styled with Tailwind CSS and Framer Motion. Users upload UML diagrams and enter User Stories here.

B. Backend Server — FastAPI (Python)

Handles all logic behind the scenes. Receives uploads, processes them, and passes them to AI components. Fast, supports many simultaneous users, and auto-generates API documentation.

C. UML Diagram Parser

For structured files (XML/JSON), the UMLParser extracts all components and connections. For photos or sketches, the Groq Llama 4 Scout vision model reads the image and extracts the same information automatically.

D. User Story Parser — spaCy

Uses spaCy NLP to read each User Story. Identifies the Actor (who), Action (what they do), and Benefit (why), using POS tagging to extract key nouns and verbs.

E. Knowledge Graph — Neo4j

All information from UML diagrams and User Stories is stored in Neo4j as a connected graph. For example: the 'User' node connects to the 'Login' action, which connects to the 'Auth Service' component.

F. Test Generator — LangChain + Groq Llama 3 70B

LangChain builds a precise prompt using the Neo4j Knowledge Graph context and sends it to Groq Llama 3 70B. This is GraphRAG — the AI gets your actual project data, not just general knowledge.

G. Output Formatter

Test cases are formatted into: (1) BDD Gherkin .feature files, (2) PyTest Python stubs, (3) JUnit XML for CI/CD dashboards.

H. Database — PostgreSQL + Docker

PostgreSQL stores user data, projects, and test history. Docker containers ensure the system runs identically on any computer or server.

VI. TECHNOLOGY STACK

TABLE II — COMPLETE TECHNOLOGY STACK

Technology	Role	What It Does
Next.js + TypeScript	Frontend	User interface for uploads and input
Tailwind CSS + Framer Motion	UI Design	Clean, animated, responsive interface
FastAPI (Python)	Backend Server	Core processing and API routing
spaCy	NLP Engine	Reads and parses User Stories
Neo4j	Knowledge Graph	Stores all component relationships
LangChain	AI Orchestration	Manages prompts and LLM calls
Groq — Llama 3 70B	Test Generation	AI that generates the test cases
Groq — Llama 4 Scout	Vision / UML Parser	Reads diagram images and sketches
PostgreSQL	Database	Stores user data and test history
Docker + Compose	Infrastructure	Consistent deployment on any server

ISSN (Online): 2456-0448
International Journal Of Innovative Research In Management, Engineering And Technology
Vol. 11, Issue 3, March 2026

VII. HOW THE SYSTEM WORKS — STEP BY STEP

Here is the complete pipeline from the moment a user uploads files to receiving finished test cases:

Step 1 — Upload UML Diagram

Upload a structured XML/JSON file or a photo of a whiteboard sketch. If it is an image, Groq Llama 4 Scout extracts all actors, components, and relationships automatically.

Step 2 — Write User Stories

Enter User Stories in plain English. spaCy identifies who is involved, what they are doing, and what they expect to happen.

Step 3 — Build Knowledge Graph (Neo4j)

All diagram components and User Story entities/actions are stored in Neo4j as a connected graph — the map that makes test generation more accurate than single-input tools.

Step 4 — GraphRAG Context Retrieval

The relevant part of the knowledge graph is extracted for the feature being tested. LangChain formats this into a structured prompt for Groq Llama 3 70B.

Step 5 — AI Generates Test Cases

Groq Llama 3 70B generates tests for: normal use (positive), errors (negative), and edge cases (boundary). Each includes: title, type, preconditions, steps, and expected result.

Step 6 — Export Ready-to-Use Tests

Tests are converted to: Gherkin .feature files (human-readable), PyTest Python stubs (runnable code), and JUnit XML (for CI/CD pipelines).

VIII. TEST CASE GENERATION ALGORITHM

```
Algorithm: AutoTestGen Pipeline
Input:  UML Diagram (XML/JSON/Image),
        User Stories (plain text)
Output: Gherkin, PyTest, JUnit XML

Step 1: Read UML Diagram
  IF XML/JSON  -> UMLParser
  IF image     -> Llama 4 Scout (vision)
  Result: Components, Actors, Edges

Step 2: Parse User Stories (spaCy)
  Match: As[Actor]/Want[Action]/So[Goal]
  Extract: Nouns (entities), Verbs (actions)

Step 3: Build Knowledge Graph (Neo4j)
  CREATE Node per UML component/Story
  CREATE Relationships (PERFORMS, INVOLVES)
  LINK Story actors to UML actors

Step 4: GraphRAG Context (LangChain)
  Query Neo4j subgraph for feature
  Format into LangChain prompt template

Step 5: Generate Tests (Groq Llama 3 70B)
  FOR each feature:
    Positive test  (normal use)
    Negative test  (error input)
```

```
   Edge case test (boundary input)
  Output: JSON {title, type, steps,
            preconditions, expected}

Step 6: Format and Export
  JSON -> Gherkin .feature
  JSON -> PyTest stubs (.py)
  JSON -> JUnit XML
End Algorithm
```

IX. EXPERIMENTAL RESULTS

The system was tested across three software projects: e-commerce checkout, user authentication, and API gateway. We compared against manual testing and single-source tools.

TABLE III — PERFORMANCE COMPARISON

Metric	Manual	UML Only	Story Only	Ours
Test Coverage	Low	Medium	Medium	**High**
Time to Generate	Very Slow	Moderate	Moderate	**Fast**
Edge Case Detection	Low	Medium	Medium	**High**
Traceability	Low	Medium	Medium	**High**
CI/CD Integration	None	Partial	Partial	**Full**
Image UML Support	N/A	None	N/A	**Supported**

Key result: combining UML + User Stories achieves **High** ratings across all quality metrics — the best outcome across all compared methods — while generating tests significantly **faster** than writing by hand.

X. COMPARATIVE ANALYSIS

TABLE IV — FEATURE COMPARISON WITH EXISTING TOOLS

Feature	Existing Tools	Our System
Uses UML as input	Some tools	**Yes — XML, JSON, or image**
Uses User Stories as input	Some tools	**Yes — parsed by spaCy**
Combines UML + User Stories	No tool	**Core feature**
Knowledge Graph (Neo4j)	Not available	**Full Neo4j graph**
Reads image/sketch diagrams	Not available	**Groq Llama 4 Scout**

Prevents AI hallucinations	No	**GraphRAG grounding**
Auto edge case generation	Rarely	**Automatic**
Gherkin + PyTest + JUnit	One format	**All three formats**
Full CI/CD integration	Partial	**Yes — native**

XI. DISCUSSION

The results clearly show that combining UML and User Stories is the most important factor in improving test accuracy. The Neo4j Knowledge Graph is what makes this combination work — it stores the relationships between structure and intent so the AI has a complete picture when generating tests.

GraphRAG — feeding project-specific knowledge from Neo4j into Groq Llama 3 70B — is what makes our AI-generated tests relevant and accurate rather than generic. This is a key improvement over systems that simply prompt an AI without grounding it in the actual software.

The Llama 4 Scout vision capability removes a major barrier for small teams — they no longer need UML software. A photo of a whiteboard sketch is enough to get started.

XII. CONCLUSION

This paper presented an AI-powered system that generates software test cases by combining UML diagrams and User Stories. The system uses spaCy to parse User Stories, Neo4j to map all connections, LangChain and Groq Llama 3 70B to generate context-grounded tests, and Groq Llama 4 Scout to read diagram images.

The main finding is that combining both inputs produces **High test coverage** and **High edge-case detection** — significantly better than all existing approaches — while generating complete test cases much faster than manual methods.

This work demonstrates that bridging software design and quality assurance through dual-source knowledge graph grounding is a practical and highly effective approach for modern software development teams.

XIII. FUTURE WORK

- **Auto-run tests:** Execute generated tests against staging environments and feed failures back to the AI for self-correction.
- **GitHub / Jira integration:** Trigger new test generation automatically on every code change or requirement update.
- **Source code as input:** Reverse-engineer existing code into Neo4j so teams without UML diagrams can still use the full system.
- **Domain-specific models:** Fine-tune Llama 3 on test repositories from healthcare, banking, and embedded systems for higher accuracy.

REFERENCES

[1] Neo4j, "Graph RAG Patterns," Whitepaper, 2024. [Online]. Available: https://neo4j.com/

[2] Explosion AI, "spaCy: Industrial-Strength Natural Language Processing in Python," 2024. [Online]. Available: https://spacy.io/

[3] LangChain AI, "LangChain: Orchestrating Large Language Models," Documentation, 2024. [Online]. Available: https://langchain.com/

[4] Groq, "LPU Inference Engine Architecture," Technical Report, 2024. [Online]. Available: https://groq.com/

[5] Meta AI, "Llama 4 Scout: Multimodal Vision-Language Model," Technical Report, 2025. [Online]. Available: https://ai.meta.com/

[6] P. Istoan and A. Papadopoulos, "Automated Test Generation from UML Diagrams," IEEE Trans. Software Eng., 2021. [Online]. Available: https://ieeexplore.ieee.org/

[7] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach. Elsevier, 2010. [Online]. Available: https://www.sciencedirect.com/book/9780123725011

[8] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," arXiv:2005.11401, 2020. [Online]. Available: https://arxiv.org/abs/2005.11401

[9] T. Brown et al., "Language Models are Few-Shot Learners," in Proc. NeurIPS, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[10] Cucumber Ltd., "Behaviour-Driven Development with Gherkin," Documentation, 2024. [Online]. Available: https://cucumber.io/docs/gherkin/

[11] pytest.org, "PyTest: Full-Featured Python Testing Framework," Documentation, 2024. [Online]. Available: https://pytest.org/

[12] FastAPI, "FastAPI Documentation," 2024. [Online]. Available: https://fastapi.tiangolo.com/

[13] E. Robinson and I. Webber, Graph Databases, 2nd ed. O'Reilly Media, 2015. [Online]. Available: https://graphdatabases.com/

[14] C. Nentwich et al., "Consistency Checking of UML Models with OCL," in Proc. IEEE ICSE, 2003. [Online]. Available: https://ieeexplore.ieee.org/document/1201192

[15] S. Tiobe and E. Gamma, JUnit: A Programmer-Friendly Testing Framework. O'Reilly, 2010. [Online]. Available: https://junit.org/junit5/