happiest minds
AI FIRST. AGILE ALWAYS.

RETHINKING THE SOFTWARE TEST LIFECYCLE

THROUGH A GENERATIVE AI LENS

CONTENTS

01

EXECUTIVE SUMMARY

Testing has become one of the most time-consuming and labor-intensive parts of the delivery lifecycle as software systems get more complicated.

With the rapid development of Generative AI, there is a huge opportunity to rethink/relook at how test development is performed. Instead of treating requirements as static documents that require manual interpretation at every stage, Generative AI enables them to serve as dynamic inputs that can directly drive test design and automation.

Motivated by this transition, the purpose of this endeavour is to explore how Large Language Models (LLMs), accompanying technologies such as Lang Chain and GitHub Copilot, may be implemented across the Software Test Lifecycle to reduce manual work, increase consistency, and expedite test readiness. The strategy focuses on using Business Requirement Documents and scenario descriptions as a single source of truth.

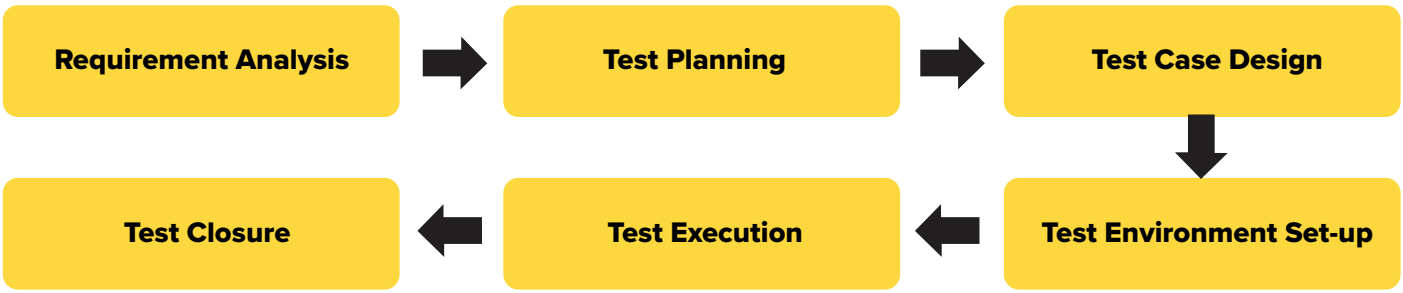
By leveraging Gen AI, these inputs are transformed into structured test scenarios, detailed test cases, automation scripts, and reusable framework components. AI support is further applied to review generated test artifacts and analyze test results, enabling a more complete and intelligent testing workflow.

This article presents the approach, architecture, and key learnings, which illustrate how AI can shift testing from a manual, document-driven activity to an intelligent, AI-assisted engineering, making faster delivery, higher quality, and long-term scalability across domains.

02

INTRODUCTION

The Software Testing Life Cycle (STLC) is a systematic procedure for testing software applications to ensure software quality goals are met.



The emergence of Artificial Intelligence introduces a new pattern for software testing. Most traditional automation tools work the same way they always have: they rely on rigid scripts and predefined rules, and they follow those instructions like a robot on a factory line.

AI is different. Powered by large language models (LLMs), it can read natural language — the way real people write and talk — and then turn that into useful stuff like test scenarios, test cases, or even working code. That difference changes how we approach test creation throughout the Software Testing Life Cycle (STLC).

Instead of treating requirement documents like dusty PDFs that must be manually decoded every time, AI lets those documents become living, active participants in the testing process. The requirements stop being just static inputs—they start helping generate tests with traceability.

This whitepaper walks through how AI fits into every major stage of the Software Testing Life Cycle — from making sense of requirements to smart test design, all the way through generating automation scripts, building frameworks, validating the outputs, and even analyzing test results. We'll look at real ways to bring tools like large language models, LangChain, and GitHub Copilot into your day-to-day testing workflows, reducing repetitive manual effort and accelerating automation readiness.

The complete process involves multiple stages starting from

1. **Requirement analysis**
2. **Test Scenario generation**
3. **Test case generation**
4. **Test Execution reports analysis**

03

PROBLEM STATEMENT

TRADITIONAL TESTING APPROACH LIMITATIONS

With all the testing frameworks, early critical parts of the STLC are being done as manual in most cases. Things like figuring out what the requirements actually mean, coming up with test scenarios, writing detailed test cases, and getting everything ready for automation... most of that still falls on human shoulders. It takes forever, it's exhausting, and honestly, it's way too easy to mess up.

3.1

The Endless Loop of Manual Requirement Interpretation

In the old-school approach, your Business Requirement Documents (BRDs), functional specs, or user stories are basically treated like ancient scrolls. Someone has to sit down, read them line by line, interpret what they really mean, and then manually turn that into test scenarios and cases.

The trouble is that every tester reads things a little differently. One person might catch a subtle edge case; another might completely miss it. The result? Inconsistent coverage—some features get over-tested; others barely touched—and gaps that only show up later when bugs slip into production. It's not anyone's fault; it's just human nature. But it costs time, money, and a lot of frustration.

3.2

Test Design

Test scenario identification and test cases are among the most time-consuming phases of the STLC. Engineers manually analyze requirements to identify positive, negative, and boundary conditions, often repeating similar test logic across different projects or releases. This repetitive effort not only slows down test readiness but also diverts skilled testers away from higher-value activities such as exploratory testing and quality analysis.

Identifying scenarios and writing test cases is one of the most time-consuming parts of testing. You're combing through requirements, thinking through happy paths, error conditions, boundary values, negative flows, and often doing the exact same activities, done on the last project or the last release.

3.3

Test Result Analysis That Tells You... Not Much

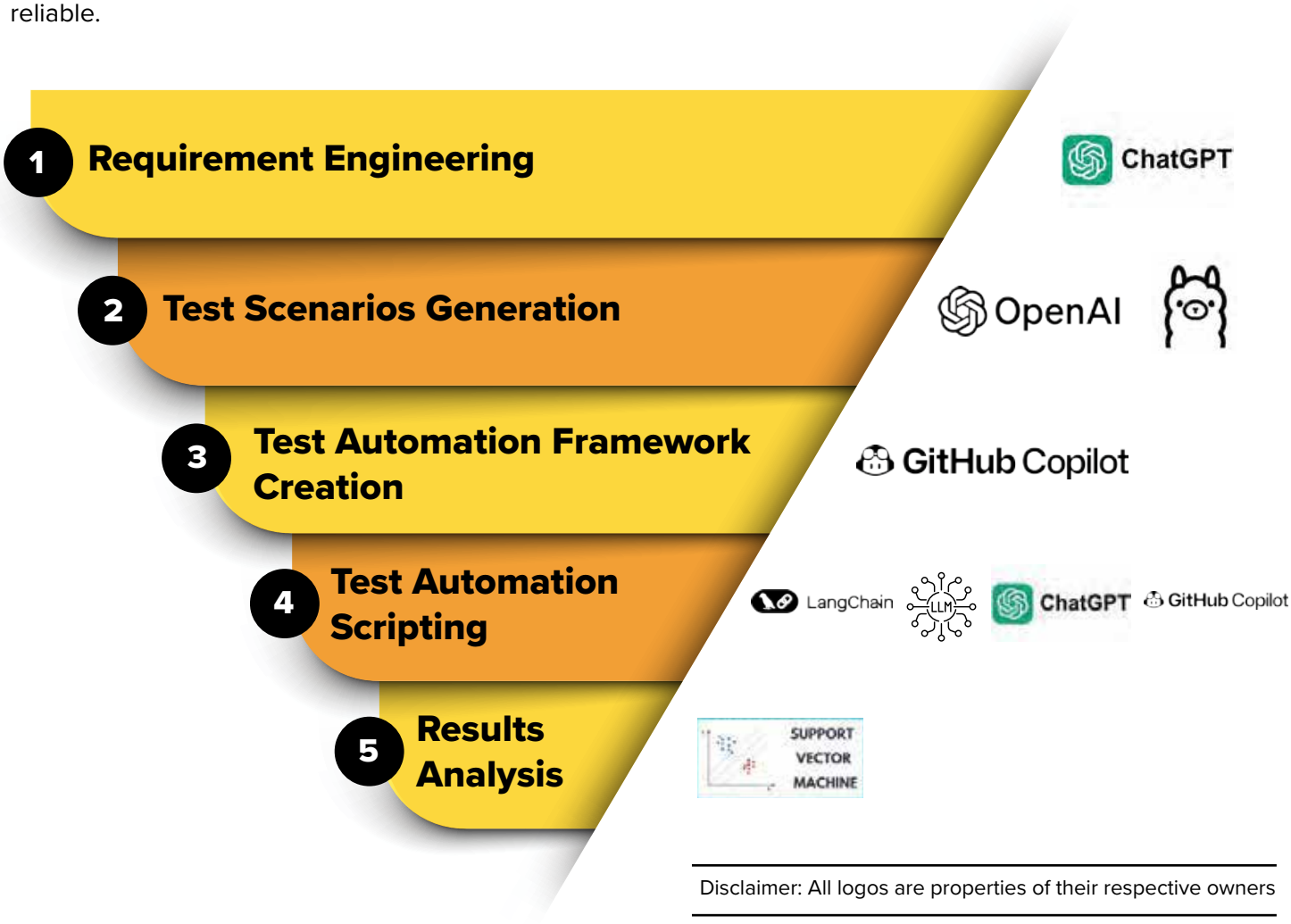
When tests finally run, most traditional reports boil everything down to green or red: pass or fail. That's useful, sure, but it doesn't tell the full story. Is a flaky test uncovering real instability, or is it just environmental noise? How healthy is this feature over time? Where are the hidden risks building up?

Digging into those insights usually means manually sifting through mountains of logs and reports—after the fact. It's reactive instead of proactive, and by the time you spot a pattern, you're often already fixing issues instead of preventing them.

04

HIGH LEVEL ARCHITECTURE

It's built as a layered, modular setup where each stage feeds into the next, keeping everything traceable from the original requirements right through to execution and analysis. The key here is to merge large language models (LLMs) with smart prompting and automated code completion, making the whole process scalable and reliable.



Disclaimer: All logos are properties of their respective owners

Requirement Analysis Phase

The major refinement occurs here; it forms the basis of all the further stages in the entire AI-driven STLC. We take all the usual inputs:

Business Requirement Documents (BRDs)	**Requirement Analysis Documents**	**Scenario descriptions**	**Specifications (JSON, Swagger, or plain text)**

The goal? Dig through these often-lengthy documents, strip out the fluff—the extra wording and non-technical bits—and distill it down to the core technical essence. We feed this into powerful LLMs (such as those from OpenAI) to automatically refine it. The output is a clean, accurate version ready for further steps.

Generating Test Scenarios

With that refined requirements document in hand, we move on to creating test scenarios. Every key piece of the cleaned-up BRD gets sent to the LLM via API calls. By crafting the right prompts, we guide the model to expand each requirement into comprehensive scenarios that cover:

- **Positive Test Cases**
- **Negative Test Cases**
- **Boundary and Edge-condition Test Cases**

Traceability between requirements, scenarios, and test cases is preserved through identifiers and metadata.

Detailed Test Cases Generation

Now we take both the refined requirements and the newly generated scenarios as our reference point to create actual test cases. Again, we use LLMs through exposed APIs to produce structured, ready-to-use test cases. Each one typically includes:

- **Preconditions (what needs to be set-up first)**
- **Test steps**
- **Input data (Sample input data)**
- **Expected results**

The test cases generated includes:

- Functional
- Negative and failure-path
- Boundary and edge-condition test cases

The advantage of this AI-assisted approach is in the consistency. and traceability stays intact—one scenario might map multiple test cases, but you can always follow the thread back.

Skeletal Framework Builder

Tools like GitHub Copilot act as "skeletal developers." By providing smart code completions, Copilot can automatically generate:

- **Standardized Folder Structure**
- **Configuration Files (like YAML or JSON for environments)**
- **Logging mechanisms are auto-generated to support:**
  - Debugging
  - Execution traceability

Teams save a significant amount of time and effort while avoiding common pitfalls like inconsistent structures or missing logging. This foundational stage becomes faster and more reliable, letting you focus on what matters: the tests themselves.

Test Scripter

Once the framework is done, the next step is converting test cases and scenarios into executable code. For this phase, the inputs are like:

- **Pre-Generated test cases and scenarios**
- **Generating reusable components, wrappers, and helpers**
- **Framework structure created earlier**

Using Lang Chain-enabled workflows, multiple chained prompts are applied to:

- Convert test cases into executable Python test scripts (.py format)
- Apply framework-specific conventions

Lang Chain's strength in handling chained interactions makes it ideal for producing high-quality, maintainable scripts without endless manual coding

Intelligent Test Execution Report Analysis

The final stage closes the loop: analyzing results after running your test suites. Instead of manually sifting through reports, feed them into a machine-learning model for deeper insights.

One effective approach is a logistic regression model built with Python libraries (like scikit-learn). Tag each test case with relevant features (e.g., module, type, or historical behavior), then use both current and historical execution data to compute:

Stability metrics for individual test cases (e.g., flakiness scores)

Feature robustness indicators

Trend-based quality insights

This enables teams to:

- Identify flaky or unstable test cases
- Understand feature-level quality trends
- Make data-driven decisions for regression planning

		INPUT		
Business Requirements, Product Specs, Design, Custom Scenarios / Swagger File	For API: Swagger.JSON For Industrial: Requirement Analysis Document	Test Cases / Test Scenarios Excel Sheet	Test Scripts	Detailed Test Reports
		Test Phases		
Requirement Engineering	**Test Case / Scenario Generation**	**Test Scripting**	**Test Execution**	**Report Analysis**
		OUTPUT		
For API: Swagger.JSON For Industrial: Requirement Analysis Document	Test Scenarios Excel Sheet Test Cases in Excel Sheet	Python oriented OPCUA/REST API test code generation using AI models Test Code review by additional AI Models/tools Test Code generated is used by copilot for Framework Structuring	Test Reports	Test Case Stability Index Test Classification for Future regression runs Feature Stability Index

05

USE CASE

GEN AI FOR INDUSTRIAL 4.0 OPC UA

It's built as a layered, modular setup where each stage feeds into the next, keeping everything traceable from the original requirements right through to execution and analysis. The key here is to merge large language models (LLMs) with smart prompting and automated code completion, making the whole process scalable and reliable.

5.1 Objective

Explore the usage of the Gen AI tools, Open API and other AI models for requirement elaboration, test case generation and Test Automation Support in Industrial Use cases

Increase in test engineer productivity

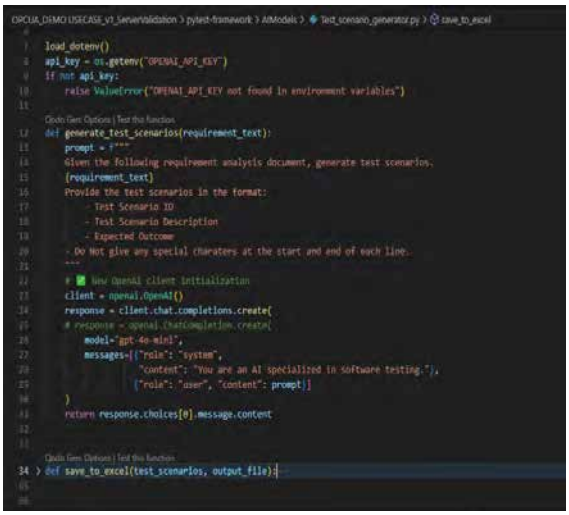
5.2 Planned Outcome

List of all tools, AI models and other options that can be used at various stages of STLC

Readiness to try out some of the existing projects as a sidebar to evaluate further

Input: Requirement Document/Requirement Scenarios

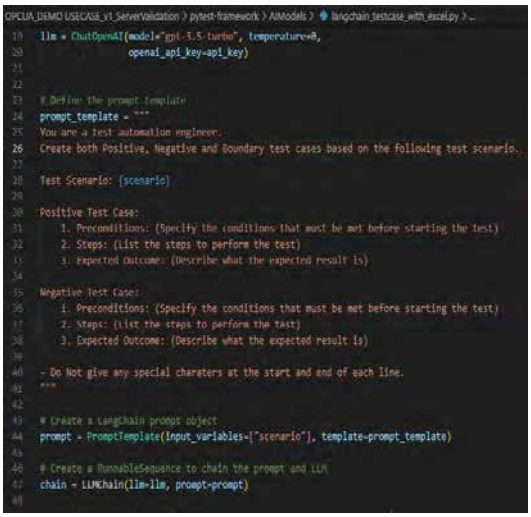
5.3 Outcomes & Screenshots



```
load_dotenv()
api_key = os.getenv("OPENAI_API_KEY")
if not api_key:
    raise ValueError("OPENAI_API_KEY not found in environment variables")

def generate_test_scenarios(requirement_text):
    prompt = f"""
    Given the following requirement analysis document, generate test scenarios.
    {requirement_text}
    Provide the test scenarios in the format:
        - Test Scenario ID
        - Test Scenario Description
        - Expected Outcome
    - Do Not give any special charaters at the start and end of each line.
    """
    # ✅ New OpenAI client initialization
    client = openai.OpenAI()
    response = client.chat.completions.create(
    # response = openai.ChatCompletion.create(
        model="gpt-4o-mini",
        messages=[{"role": "system",
                   "content": "You are an AI specialized in software testing."},
                  {"role": "user", "content": prompt}]
    )
    return response.choices[0].message.content


def save_to_excel(test_scenarios, output_file):
```



```
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0,
                 openai_api_key=api_key)


# Define the prompt template
prompt_template = """
You are a test automation engineer.
Create both Positive, Negative and Boundary test cases based on the following test scenario.

Test Scenario: {scenario}

Positive Test Case:
    1. Preconditions: (Specify the conditions that must be met before starting the test)
    2. Steps: (List the steps to perform the test)
    3. Expected Outcome: (Describe what the expected result is)

Negative Test Case:
    1. Preconditions: (Specify the conditions that must be met before starting the test)
    2. Steps: (List the steps to perform the test)
    3. Expected Outcome: (Describe what the expected result is)

- Do Not give any special charaters at the start and end of each line.
"""

# Create a LangChain prompt object
prompt = PromptTemplate(input_variables=["scenario"], template=prompt_template)

# Create a RunnableSequence to chain the prompt and LLM
chain = LLMChain(llm=llm, prompt=prompt)
```

TEST SCENARIO CREATION

Test Category	Test Scenario ID	Test Scenario Description	Expected Outcome
Functional	**TS001**	** Verify that the gateway functions as an OPCUA client and communicates seamlessly with OPCUA-compliant sensors.	The gateway successfully establishes communication with OPCUA-compliant sensors and retrieves data without errors.
Functional	**TS002**	** Verify that the gateway collects real-time data from temperature, vibration, gas, and pressure sensors.	Real-time data from all sensors is collected and displayed accurately on the dashboard.
Functional	**TS003**	** Verify that the gateway processes sensor data and sends it to the cloud using MQTT or HTTP protocols.	Sensor data is transmitted to the cloud without data loss or delays.
Functional	**TS004**	** Verify that the gateway stores sensor data locally using an embedded database when the internet connection is lost.	Sensor data is stored locally in the embedded database without any data loss during internet outages.
Functional	**TS005**	** Verify that the gateway syncs locally stored data with the cloud once the internet connection is restored.	All locally stored data is successfully synced with the cloud without duplication or loss.
Functional	**TS006**	** Verify that the system allows configurable retention policies for local storage (e.g., up to 48 hours).	The system retains locally stored data based on the configured retention policy and deletes older data as per the policy.
Functional	**TS007**	** Verify that the system supports rule-based automation for control mechanisms.	Rule-based automation is executed as per the defined rules without errors or delays.
Functional	**TS008**	** Verify that users can configure real-time alerts for critical conditions via email, SMS, and push notifications.	Alerts are successfully configured and delivered via the selected communication channels.
Functional	**TS013**	** Verify that event-driven alerts are triggered based on business rules (e.g., unexpected pressure drop in pipelines).	Event-driven alerts are triggered accurately based on the defined business rules.
Functional	**TS014**	** Verify that escalation alerts are triggered if the primary responder does not acknowledge an alert within the set time.	Escalation alerts are sent to the next level of responders as per the configured escalation policy.

TEST CASE CREATOR

**** Verify that the gateway functions as an OPCUA client and communicates seamlessly with OPCUA-compliant sensors and systems.**

Test Scenario ID
TS001

Test Case ID
TC001

Test Case Name
Verify gateway functions as an OPCUA client and communicates with OPCUA-compliant sensors

Test Type
Automation

Preconditions

The gateway device is powered on and connected to the network.

OPCUA-compliant sensors (e.g., temperature, vibration, gas, pressure) are installed and operational.

The gateway is configured with the correct OPCUA server endpoint.

The OPCUA server is running and accessible.

Expected Outcome

The gateway successfully establishes a connection with the OPCUA server.

The gateway receives and processes real-time data from all connected OPCUA-compliant sensors.

The gateway logs errors or warnings when a sensor is disconnected.

The gateway resumes data collection automatically when the sensor is reconnected.

Steps

1. Start the gateway device and ensure it is connected to the network.
2. Configure the gateway with the OPCUA server endpoint details (e.g., IP address, port, and security settings).
3. Initiate the OPCUA client connection from the gateway to the OPCUA server.
4. Verify that the gateway successfully establishes a connection with the OPCUA server.
5. Simulate sensor data from the OPCUA-compliant sensors (e.g., temperature, vibration, gas, pressure).
6. Verify that the gateway receives real-time data from the sensors via the OPCUA protocol.
7. Check the logs on the gateway to confirm successful data transmission from the sensors.
8. Disconnect one sensor and verify that the gateway logs an error or warning indicating the sensor is unavailable.
9. Reconnect the sensor and verify that the gateway resumes data collection without manual intervention.

TEST CASE CREATOR

**** Verify that the gateway collects real-time data from temperature, vibration, gas, and pressure sensors.**

Test Scenario ID
TS002

Test Case ID
TC001

Test Case Name
Verify real-time data collection from temperature sensors

Test Type
Automation

Preconditions

The gateway device is powered on and connected to the network.

Temperature sensors are properly installed and configured to communicate with the gateway.

The gateway is configured to collect data from temperature sensors.

Steps

1. Simulate temperature sensor data using an OPCUA-compliant simulator.
2. Ensure the gateway is connected to the simulator.
3. Start the data collection process on the gateway.
4. Monitor the gateway logs or dashboard to verify that temperature data is being received in real-time.
5. Validate the data format and values against the simulated input.

Expected Outcome

The gateway collects and displays real-time temperature data from the sensors.

The data format and values match the simulated input.

**** Verify that the gateway collects real-time data from temperature, vibration, gas, and pressure sensors.**

Test Scenario ID
TS002

Test Case ID
TC002

Test Case Name
Verify real-time data collection from vibration sensors

Test Type
Automation

Preconditions

The gateway device is powered on and connected to the network.

Vibration sensors are properly installed and configured to communicate with the gateway.

The gateway is configured to collect data from vibration sensors.

Steps

1. Simulate vibration sensor data using an OPCUA-compliant simulator.
2. Ensure the gateway is connected to the simulator.
3. Start the data collection process on the gateway.
4. Monitor the gateway logs or dashboard to verify that vibration data is being received in real-time.
5. Validate the data format and values against the simulated input.

Expected Outcome

The gateway collects and displays real-time vibration data from the sensors.

The data format and values match the simulated input.

TEST REPORT ANALYZER

```
Test Cases Grouped by Predicted Robustness Level (SVM):
  Predicted Robustness Level Features List
0                     Robust    F1, F2, F3
0                       Good            F1
0                      Flaky            F3

Full Dataset:
  Test Case ID      Features  ... Robustness Encoded  Predicted Robustness Level
0          TC1  [F3, F1, F2]  ...                   2                      Robust
1          TC2      [F2, F1]  ...                   2                      Robust
2          TC3          [F3]  ...                   0                       Flaky
3          TC4          [F1]  ...                   1                        Good
4          TC5      [F2, F3]  ...                   2                      Robust
```

06

CONCLUSION

The adoption of Generative AI across the Software Test Lifecycle presents a meaningful shift in how testing activities are performed, however as AI models are effective at extracting relevant information from requirement documents, generating diverse test cases, and identifying patterns in test results, they can struggle with highly complex business logic or domain-specific nuances. Similarly, auto-generated test cases and scripts may require refinement to remove redundancy or to better reflect real-world usage scenarios.

The most effective outcomes were observed when AI was positioned as a co-pilot rather than an autonomous system. Human-in-the-loop validation remains critical—particularly during requirement interpretation, test case review, and defect analysis—to ensure correctness, contextual understanding, and quality assurance. This balanced approach helps mitigate risks such as over-reliance on AI, misinterpretation of failures, or gaps in validation.

Generative AI is not a replacement for engineering judgment, but a powerful enabler when used with the right level of human oversight

ABOUT THE AUTHOR

NIKHIL NAGENDRA

Test Architect I Test Automation Specialist
Embedded & Industrial Systems

With 11 years of experience in software testing, Nikhil Nagendra is a skilled Test Architect specializing in Test Automation for embedded and industrial projects. Passionate about driving efficiency and precision in testing, he has worked on automating complex systems in industrial environments.

His expertise includes framework/library development across multiple [pytest, Robot, and BDD] frameworks, protocol analysis on embedded devices, CI/CD support, usage of various testing tools and multiple programming languages.

Nikhil is also exploring AI-driven test automation, aiming to leverage machine learning and intelligent automation techniques to enhance testing strategies. With a keen eye for innovation and quality, he strives to optimize testing processes and ensure robust product validation across domains.

About Happiest Minds Technologies

Happiest Minds Technologies Limited (BSE, NSE: HAPPSTMNDS) is an AI First, customer-centric digital engineering company committed to delivering 'Happiest People . Happiest Customers'. With an integrated approach that spans from chip to cloud, Happiest Minds delivers secure and scalable solutions across product engineering, cybersecurity, analytics , and automation platforms. Happiest Minds brings purpose and precision to every engagement, helping enterprises solve complex business challenges and fast-track their digital evolution across industry sectors such as Banking, Financial Services & Insurance (BFSI), EdTech, Healthcare & Life Sciences, Hi-Tech and Media & Entertainment, Industrial, Manufacturing, Energy & Utilities, and Retail, CPG & Logistics.

Happiest Minds has been honored by both the Golden Peacock Awards and the Institute of Company Secretaries of India (ICSI) for its exemplary Corporate Governance practices. Guided by its mission of 'Happiest People . Happiest Customers' and consistently recognized as a great place to work, Happiest Minds is headquartered in Bengaluru, India, with a global presence across the Americas, UK, Europe, Australia, the Middle East, Africa, and Asia.

happiest minds
AI FIRST. AGILE ALWAYS.

For more information, write to us at
business@happiestminds.com