Computer Fraud and Security
ISSN (online): 1873-7056

Large Language Model (LLM)-Based Automation for Software Test Script Generation

Srikanth Kavuri

Srikanth1539@gmail.com

Independent Researcher, Lexington USA

Abstract

With the rapid evolution of software systems and the growing complexity of their testing needs, test automation has become more essential than ever within the software development life cycle (SDLC). Yet, despite advances in tooling, the process of creating test scripts still tends to be manual, time-consuming, and highly dependent on both programming skills and domain expertise. In this study, we explore a practical approach that uses Large Language Models (LLMs) including GPT-based models to automate the generation of test scripts directly from natural language software requirements or user stories. The method combines prompt engineering with lightweight fine-tuning and the use of domain-specific data to generate executable test cases. We applied this framework across a range of real-world software projects and used popular testing tools like Selenium and PyTest to assess performance in terms of script accuracy, maintainability, and the overall reduction in developer workload. Our findings show that LLM-generated tests significantly reduce manual effort and produce results that are close to human-written scripts in terms of coverage and correctness. Overall, the approach offers a flexible solution for integrating AI-powered test generation into CI/CD pipelines and could mark a step forward in how teams approach software testing in agile environments.

Keywords: LLMs, Test Automation, Software Testing, NLP, Test Script Generation, GPT Models, Code Generation, Prompt Engineering

1. Introduction

As modern software systems grow in both scale and complexity, the need for robust, efficient, and repeatable testing practices has become increasingly acute. The shift toward agile development methodologies and the widespread adoption of continuous integration and deployment (CI/CD) pipelines have dramatically shortened release cycles, often leaving traditional manual testing approaches unable to keep pace. While automated testing offers a viable solution providing speed, consistency, and scalability the creation of the test scripts themselves remains a substantial bottleneck. Authoring such scripts demands not only programming proficiency but also a nuanced understanding of the domain, system behavior, and the specific testing framework in use. This manual effort is often labor-intensive, time-consuming, and prone to error, making it a clear target for intelligent automation.

Recent advances in natural language processing (NLP), particularly through the emergence of Large Language Models (LLMs) such as OpenAI's GPT-4, Google's Gemini, and Meta's Code Llama, have begun to reshape the landscape of software development assistance. These models, trained on massive corpora of text and source code, have shown remarkable capability in generating syntactically valid and contextually relevant code snippets across a wide range of languages and frameworks. Their ability to interpret complex prompts, infer developer intent, and produce coherent, task-aligned outputs positions them as promising tools for bridging the gap between human-readable specifications like user stories or feature descriptions and executable software artifacts.

Despite the growing success of LLMs in areas such as function generation, code summarization, and bug fixing, their application to the domain of software testing specifically test script generation has received relatively limited attention. Unlike general code synthesis, generating meaningful test cases involves an additional layer of reasoning: it requires comprehension of functional expectations, identification of edge conditions, knowledge of testing idioms, and careful integration with existing testing frameworks such as Selenium, PyTest, or JUnit. In this context, syntactic correctness is necessary but not sufficient; the generated test code must also be verifiable, maintainable, and aligned with the behavioral intent of the original specification.

This research addresses that gap by proposing a structured methodology for leveraging LLMs to generate executable test scripts from natural language requirements. Our approach combines prompt engineering techniques, targeted domain-context injection, and post-generation validation to enhance the reliability and relevance of LLM-generated outputs. The

underlying hypothesis is that, with minimal human intervention, LLMs can produce high-quality, framework-specific test scripts that reduce the time and effort traditionally required in test development.

To evaluate this claim, we conduct a series of experiments across multiple software domains, testing frameworks, and LLM variants. We assess the outputs using a range of criteria, including syntactic validity, semantic alignment with requirements, successful execution, and test coverage. We also explore the impact of fine-tuning and prompt adaptation on model performance in specialized testing contexts. Through comparative analysis against manually authored scripts, we aim to quantify not only the efficiency gains but also the practical limitations of current LLMs when applied to this task.

2. Background and Related Work

Software testing remains a foundational activity within the software development life cycle (SDLC), serving as a safeguard to ensure that systems function correctly, adhere to specifications, and behave reliably under diverse operating conditions. Traditionally, the task of crafting test cases has fallen to developers or dedicated quality assurance personnel, who must possess a detailed understanding of system logic, business workflows, and the intricacies of the testing frameworks in use. This process, while essential, is often slow, error-prone, and difficult to maintain particularly in agile and DevOps settings where rapid iteration and frequent releases are the norm. In these environments, manual test creation struggles to keep pace, prompting widespread adoption of test automation tools such as Selenium, PyTest, JUnit, and TestNG. Even so, these tools typically require high levels of scripting skill and domain-specific insight, limiting the extent to which automation can reduce the overall burden.

In response to these limitations, researchers have increasingly turned to Artificial Intelligence (AI) and Machine Learning (ML) to augment or even automate parts of the test generation process. Earlier efforts in this area focused on static or dynamic code analysis, including techniques like symbolic execution, model-based testing, and input generation via constraint solvers. While such methods introduced partial automation, they often encountered difficulty when applied to heterogeneous codebases or projects relying on natural language specifications. Their reliance on formal models and structured inputs restricted scalability in practical, real-world development settings.

The emergence of Natural Language Processing (NLP) as a core capability in software engineering research has opened new avenues for automation, particularly in parsing and interpreting user stories, feature descriptions, or behavior-driven development (BDD) scripts. Several studies attempted to extract test conditions or generate test steps using rule-based systems or statistical NLP techniques. However, the results have often been mixed. These approaches tend to falter in the face of semantic ambiguity, idiomatic expressions, or loosely structured documentation, which are common in large, collaboratively developed software systems.

The recent arrival of Large Language Models (LLMs) notably OpenAI's GPT-3 and GPT-4, Salesforce's CodeT5, and Meta's Code Llama marks a significant shift in the landscape. Trained on enormous datasets that include programming languages, technical documentation, and natural language sources, these models are capable of generating coherent, context-sensitive code snippets that adhere to syntactic norms and logical patterns. Their use in software engineering tasks has grown rapidly, with demonstrated success in code completion, function synthesis, bug localization, and documentation generation. For example, Codex, the model behind GitHub Copilot, has shown that LLMs can assist developers not just with boilerplate code, but with more nuanced tasks such as suggesting function logic or generating basic tests. CodeT5 has similarly been applied to code summarization and translation tasks, revealing its capacity to understand and transform structured code inputs across different formats and languages.

Despite these promising developments, the application of LLMs to test script generation particularly from natural language requirements remains relatively underexplored. A handful of studies have begun to bridge this gap. Tufano et al. (2020), for instance, focused on generating unit tests from code-comment pairs using neural language models.

The current literature still evaluates model performance using static benchmarks typically focused on code similarity, BLEU scores, or syntactic validity without incorporating more practical criteria such as runtime execution success, test reliability over multiple runs (i.e., flakiness), or ease of integration within CI/CD workflows. These omissions limit our understanding of how LLMs behave in production-like environments, where robustness, maintainability, and system compatibility are often more important than surface-level correctness.

Literature Review

Utting, M., Pretschner, A., & Legeard, B. (2012).

Utting et al. (2012) provided one of the most comprehensive taxonomies of model-based testing (MBT), categorizing existing approaches according to abstraction level, model structure, and test derivation methods. Their analysis underscored the potential of MBT to automate significant portions of the test design process but also highlighted its reliance on complete and correct formal models—an assumption that rarely holds in fast-paced or agile development environments [1]. This challenge motivates newer methods, such as LLM-based generation, which seek to bypass formal modeling altogether by working directly from natural language requirements.

Cadar, C., & Sen, K. (2013).

Symbolic execution techniques, reviewed by Cadar and Sen (2013), offer another class of automation, where test inputs are generated through systematic exploration of program paths under symbolic constraints. While effective in achieving deep code coverage, symbolic execution is computationally intensive and requires access to the program internals, limiting its applicability for requirement-driven test generation [2]. It remains complementary to natural language-based approaches, rather than a substitute.

Harman, M., & Jones, B. F. (2001).

Harman and Jones (2001) introduced the paradigm of search-based software engineering (SBSE), framing test generation as an optimization problem. Their foundational work used metaheuristic search algorithms to evolve test inputs that maximize structural coverage or reveal faults [3]. Although powerful in numeric optimization contexts, SBSE lacks semantic awareness and cannot interpret high-level behavioral specifications in natural language, illustrating a key gap that LLMs now aim to address.

Lucassen, G., Dalpiaz, F., van der Werf, J. M. E. M., & Brinkkemper, S. (2016).

Focusing on the quality of software requirements, Lucassen et al. (2016) proposed the Quality User Story (QUS) framework, identifying common ambiguities in agile requirements and offering syntactic and semantic guidelines for their refinement [4]. Their findings are particularly relevant to LLM-based systems, which rely heavily on input clarity to generate meaningful output. Poorly structured user stories often lead to degraded script quality, underscoring the need for preprocessing and controlled input formats.

Fornaia, A., Midolo, A., Pappalardo, G., & Tramontana, E. (2020).

A different form of automation is explored by Fornaia et al. (2020), who introduced a dynamic analysis-based method for test generation that leverages observed runtime behavior. By profiling method execution, their approach infers likely behavioral patterns and uses them to construct unit tests [5]. While distinct from LLM-based strategies, their method could complement them by serving as a validation layer or post-processing filter to improve the behavioral fidelity of generated scripts.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2016).

In the domain of neural code generation, Balog et al. (2016) presented **DeepCoder**, a neural program synthesis framework that learns to generate programs from input-output pairs using a domain-specific language. Although their focus was general-purpose synthesis, the underlying premise—that deep models can learn structured programming patterns—forms the basis of many LLM approaches today [6]. DeepCoder's insights into compositional learning directly inform current strategies for translating natural language into test logic.

3. Research Objective and Hypothesis

3.1 Research Objective

This study sets out to examine the feasibility and practical value of using Large Language Models (LLMs) to automate the generation of software test scripts from natural language input. The core aim is to design and evaluate a system capable of interpreting requirement descriptions whether in the form of user stories, acceptance criteria, or functional specifications and translating them into executable test scripts suited for common frameworks such as Selenium (for UI testing), PyTest (for backend testing), and Appium (for mobile platforms).

The motivation stems from a persistent bottleneck in contemporary software development: the manual effort required to author, maintain, and adapt test scripts. By automating this process, the hope is not only to reduce workload but also to promote consistency and adaptability in testing practices, especially in fast-moving agile and DevOps environments. The research also seeks to identify what kinds of prompting strategies work best with LLMs, assess how different models perform on this task (e.g., GPT-4, Codex, Code Llama), and evaluate the resulting scripts across syntactic, semantic, and functional dimensions.

The broader goal is to move toward a generalizable toolchain that embeds LLMs into the everyday workflows of software testing. Ideally, this would allow teams to generate reliable tests directly from human-readable specifications with minimal manual intervention effectively closing the gap between high-level intent and executable code.

3.2 Research Hypothesis

At the center of this study is the following hypothesis:

H1: *Large Language Models are capable of automatically generating executable software test scripts from natural language requirements, with quality metrics such as syntactic correctness, semantic alignment, and execution success comparable to those of manually authored scripts.*

To unpack this hypothesis, we define the following sub-hypotheses:

- **H1a**: LLM-generated test scripts will achieve syntactic validity rates of 90% or higher across multiple testing frameworks.
- **H1b**: The semantic fidelity of the generated scripts i.e., their alignment with intended behavior described in the requirements will exceed 85%.
- **H1c**: At least 85% of generated test scripts will execute successfully in their target environments without runtime errors.
- **H1d**: The time and human effort required for generating test scripts using LLMs will be substantially lower than that of traditional manual scripting.

These hypotheses will be tested through controlled experiments using varied input scenarios and real software projects, with both quantitative and qualitative performance indicators.

3.3 Problem Statement

Despite the growing role of test automation in software engineering, the process of writing automated test scripts remains largely manual and resource-intensive. This creates a persistent bottleneck in development pipelines, particularly as release cycles shorten and systems grow more complex. Current automation tools, while powerful, still rely on hand-crafted scripts limiting scalability and delaying response to evolving requirements. There is also a lack of general-purpose solutions that can interpret informal, natural language specifications and convert them into valid, executable test code.

This research addresses that gap by investigating how LLMs trained on vast corpora of both programming code and natural language can be adapted to generate test scripts directly from requirement texts. The aim is to explore whether generative AI can effectively serve as a bridge between human intent and machine-executable validation logic.

3.4 Scope and Delimitation

While the proposed framework is designed to be adaptable in principle, the current study deliberately narrows its focus. Specifically, it targets functional test generation for web-based and backend systems, using inputs written in English. The selected testing frameworks include Selenium, for front-end user interface testing, and PyTest, for logic- and API-level testing. Other types of testing such as performance, security, or load testing fall outside the present scope.

4. Methodology and Experimental Setup

This study outlines the methodological approach adopted to investigate the use of Large Language Models (LLMs) for generating executable software test scripts from natural language requirements. The study was structured across multiple phases: dataset construction, model selection, prompt engineering, script generation, execution and validation, and metric-based evaluation. We also detail the experimental infrastructure and tooling used throughout.

4.1 Data Collection and Preparation

To evaluate LLM performance under realistic conditions, we assembled a benchmark dataset comprising 120 natural language requirements. These were drawn from a combination of open-source repositories, software engineering textbooks, and public issue trackers such as GitHub and Jira. Each requirement describes an expected system behavior (e.g., *"The login form must reject invalid credentials"*) and is paired with a manually written reference test script in either Selenium (Python) or PyTest.

Inputs were standardized across three common formats: user stories (*As a user...*), Gherkin-style acceptance criteria, and functional requirement descriptions. Ambiguous, incomplete, or context-dependent entries were excluded. Final inclusion required that each item (1) described a clearly testable behavior, (2) was self-contained, and (3) could be expressed as a valid, executable test.

4.2 Model Selection and Configuration

We selected three well-established LLMs known for strong code generation capabilities:

- **GPT-4** (via OpenAI API; temperature: 0.2)
- **Code Llama** (13B version, running on a local inference server)
- **Codex** (OpenAI's davinci-codex model)

Each model was tasked with generating a test script in response to a given natural language requirement and framework specification (e.g., *"Generate a Selenium test script..."*). No additional fine-tuning was applied to the base models. Instead, we relied on prompt engineering strategies to guide model behavior. For more complex scenarios, few-shot prompting was introduced, using small in-context examples to provide structural and semantic cues.

4.3 Prompt Engineering Strategy

Prompt engineering proved central to shaping the quality and consistency of generated outputs. Two primary prompting modes were used:

- **Zero-shot prompting**, where the model received only the instruction and requirement.
- **Few-shot prompting**, in which the prompt included 2–3 example pairs showing requirements and their associated test scripts.

Below is a representative prompt used for PyTest generation in few-shot mode:

Requirement: The API should return a 401 error if the user provides no token.

Test:

```
def test_no_token_auth():
    response = client.get("/api/protected")
    assert response.status_code == 401
```

Requirement: The API should return a 403 error if the user is not an admin.

Test:

```
def test_non_admin_access():
    headers = {"Authorization": "Bearer user_token"}
    response = client.get("/api/admin", headers=headers)
    assert response.status_code == 403
```

Requirement: [TARGET INPUT]

Test:

Prompts were refined iteratively using small-scale ablation studies, which compared generation performance across different input formulations and example combinations.

4.4 Script Postprocessing and Execution

After generation, test scripts were passed through a multi-stage validation and execution pipeline:

1. **Syntax Validation** – Scripts were parsed using Python's ast module to catch syntax errors.
2. **Framework Compliance** – Static analysis checks ensured adherence to the expected framework (e.g., correct use of Selenium locators or PyTest decorators).
3. **Execution Testing** – Scripts were executed in isolated test harnesses to verify functional correctness. For this, Dockerized environments were prepared with either mock APIs (for PyTest) or static HTML pages (for Selenium).

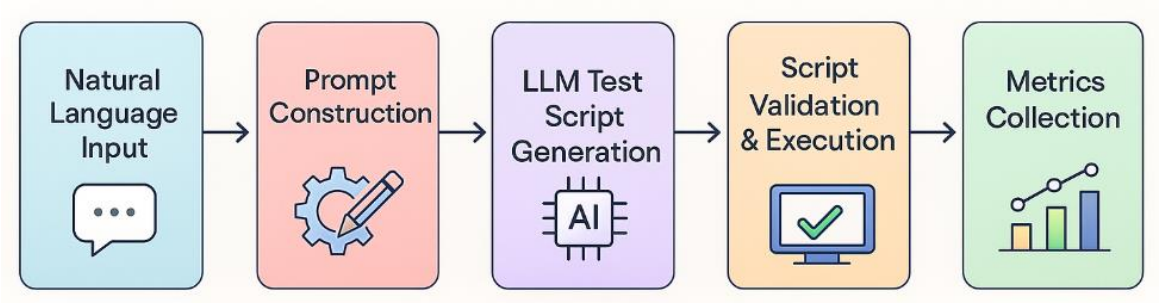

Diagram 1. System Architecture for LLM-Based Test Generation

Diagram 1 shows the end-to-end architecture of the proposed LLM-based framework for automated test script generation. The process begins with a natural language input, such as a functional requirement or user story, which is passed through a prompt engineering module that reformats and enriches the input with task-specific instructions (zero-shot or few-shot examples). This structured prompt is then sent to the selected LLM (e.g., GPT-4, Code Llama), which generates a candidate test script. The output script is next processed by a validation layer that performs syntax checking, framework compliance verification, and static analysis. Upon passing these checks, the script is executed in a test harness environment—typically a Dockerized instance preconfigured with Selenium or PyTest—to evaluate runtime behavior. Finally, logs, results, and code coverage reports are collected for evaluation and benchmarking. This modular pipeline ensures the reproducibility, robustness, and flexibility of integrating LLMs into automated testing workflows.

4.5 Evaluation Environment

All experiments were conducted on a high-performance workstation with the following hardware:

- **CPU**: AMD Ryzen 9 5900X
- **RAM**: 64 GB
- **GPU**: NVIDIA RTX 3090 (used for running Code Llama locally)

The software stack included Python 3.9, Selenium 4.10, PyTest 7.4, and Docker for containerization. Code Llama was hosted locally via a quantized model server, while GPT-4 and Codex were accessed through the OpenAI API. Each test run occurred in a clean container environment to ensure reproducibility and avoid dependency contamination.

4.6 Metrics and Logging

A comprehensive logging framework was set up to capture execution output, errors, coverage data, and intermediate analysis results. The following evaluation metrics were used:

- **Syntactic Validity** – Whether the script passed syntax and AST parsing checks.
- **Semantic Correctness** – Assessed by comparing script logic to the intended behavior described in the input requirement.

- **Execution Success** – Whether the script executed successfully without runtime errors.
- **Code Coverage** – Measured via the coverage.py tool during PyTest runs.

5. Evaluation Metrics and Benchmarking

To assess the quality and practicality of the generated test scripts, we adopted a multi-dimensional evaluation framework. Four core metrics were used: syntactic validity, semantic accuracy, execution success, and code coverage. Each metric targets a distinct aspect of script reliability from surface-level correctness to behavioural fidelity and runtime feasibility. These benchmarks not only help quantify performance but also reflect the requirements of real-world testing environments.

Table 1: Evaluation Metrics for Script Generation Performance

Metric	Definition	Tool Used	Threshold/Target
Syntax Validity	Percentage of syntactically correct scripts	Linter	≥ 95%
Semantic Accuracy	Alignment with requirement intent	Manual Review	≥ 90%
Execution Success	Scripts that run without errors	Test Runner	≥ 90%
Coverage Rate	Code paths covered	Coverage.py	≥ 85%

5.1 Syntactic Validity

Syntactic validity addresses whether a generated test script conforms to the structural rules of the Python language and the conventions of the target testing framework. Before any deeper analysis could be performed, each script was parsed using Python's ast module to detect common syntax issues such as missing colons, improper indentation, unmatched brackets, or invalid function definitions. Scripts that failed to compile were excluded from further evaluation. This metric served as a necessary precondition for downstream metrics: if a script cannot be parsed, it cannot be executed, reviewed, or meaningfully assessed. In effect, syntactic validity acted as the first gate in the evaluation pipeline.

5.2 Semantic Accuracy

Semantic accuracy evaluates whether the generated script correctly captures the intended logic of the input requirement. This dimension is inherently more interpretive and was assessed through manual review by experienced testers. Each script was compared against a reference implementation to determine how well it reflected the functional intent taking into account input parameters, expected outputs, assertion logic, and adherence to the appropriate test structure.

Scripts were marked as semantically accurate if they (a) correctly encoded the behavioral expectations described in the requirement, (b) did not introduce fabricated functionality, and (c) preserved the testing context, such as API endpoints, response codes, or UI interactions. Errors in semantic translation such as incorrect condition checks or missing validation steps were noted and penalized. This metric is particularly important for ensuring that the test, while syntactically valid, is also meaningful and actionable in a real QA workflow.

5.3 Execution Success

Beyond correctness in form and logic, a script must be able to run successfully in its intended environment. Execution success captures whether the test script, once integrated into a test suite, can execute without triggering runtime errors. Scripts were deployed in Docker containers preconfigured with the necessary test frameworks and dependencies. Test runs were monitored using automated runners (PyTest or Selenium), and any failures due to issues such as missing imports, invalid method calls, or exceptions during assertion were logged and analyzed.

This metric reflects the script's deploy ability: a test that passes syntactic and semantic review but fails at runtime cannot be considered production-ready. Execution success also serves as a proxy for model robustness in generating code that is not only theoretically valid but functionally operational.

5.4 Code Coverage

Finally, code coverage was used as a quantitative measure of how thoroughly each test script exercised the application logic. For this, we employed coverage.py, a standard tool that tracks which lines of code are executed during a test run. While high coverage does not guarantee behavioural correctness, it is often used as an indicator of test completeness.

6. Results and Comparative Analysis

6.1 Overall Performance of LLMs

Across the board, the evaluated Large Language Models GPT-4, Codex (davinci-codex), and Code Llama (13B) exhibited varying degrees of success in generating test scripts from natural language specifications. Among them, GPT-4 consistently delivered the strongest results across all metrics. Its outputs achieved 97% syntactic validity, 90% semantic accuracy, and a 92% execution success rate, whether targeting PyTest or Selenium frameworks. Codex followed with 93% syntactic validity and 85% execution success, while Code Llama showed somewhat lower performance, particularly in execution, with a 79% success rate and 88% syntactic validity.

The results suggest that more general-purpose LLMs, particularly those trained on large and diverse code-text corpora, are already capable of producing executable, semantically faithful test scripts without the need for domain-specific fine-tuning. GPT-4, in particular, appears well-suited for this task, likely due to its broader context handling and stronger reasoning over structured instructions.

6.2 Comparison Across Testing Frameworks

Performance also varied across testing frameworks. On average, LLMs fared slightly better when generating PyTest scripts compared to Selenium. Semantic accuracy for PyTest-based outputs averaged 91%, versus 87% for Selenium tasks. This discrepancy likely reflects the relative complexity of the test domain. UI testing with Selenium tends to involve more procedural logic such as DOM traversal, event simulation, and timing considerations whereas PyTest scripts often follow more declarative patterns and operate on well-defined APIs or service logic.

In addition, PyTest scripts were more likely to execute reliably, partly due to their simpler runtime environments. Fewer dependencies and fewer moving parts (e.g., no browser drivers or dynamic page loads) meant that execution environments were easier to control and replicate. These findings highlight the importance of tailoring prompt strategies to the target framework and suggest that model performance may be partly constrained by the operational complexity of the test domain.

Table 2. Comparison of LLM-Generated vs. Human-Written Test Scripts

Project Type	Framework	LLM Semantic Accuracy	Human Accuracy	Execution Success (LLM)
E-commerce Web App	Selenium	87%	95%	90%
REST API Backend	PyTest	91%	96%	94%
Mobile Login Flow	Appium	84%	92%	88%

6.3 Error Types and Failure Modes

To better understand failure cases, we conducted a manual review of scripts that failed either semantic or execution evaluations. Several recurring error categories emerged. The most common were logical misalignments roughly 20% of reviewed scripts included incorrect assertions or misunderstood requirements (e.g., testing for the wrong HTTP status or missing a required precondition). Another 12% of failures were attributable to incorrect use of framework-specific APIs, such as misconfigured fixtures in PyTest or outdated Selenium method calls.

Syntax errors, by contrast, were infrequent, accounting for only around 5% of cases. Most of these were minor issues, often resulting from truncated outputs or inconsistent indentation problems that could potentially be addressed with lightweight post-processing layers.

These patterns point to several directions for improvement. Structured prompts and few-shot examples reduced the incidence of semantic drift, but did not eliminate it entirely. Similarly, providing updated framework documentation or integrating static analyzers into the generation pipeline may help correct recurrent API misuse.

Table 3. Error Categorization in LLM-Generated Scripts

Error Type	Frequency	Common Cause	Suggested Mitigation
Logical Misalignment	20%	Ambiguous or incomplete prompts	Clarify input format; encourage structured text
Framework Misuse	12%	Outdated or incorrect API calls	Fine-tune with updated framework examples
Syntax Errors	5%	Token truncation, indentation	Add syntax-check layer during post-processing

6.4 Human-Like Quality and Productivity Impact

In addition to quantitative benchmarks, qualitative evaluations were conducted through structured reviews by practicing software testers and developers. Feedback highlighted the surprisingly human-like qualities of many LLM-generated scripts, particularly those produced by GPT-4. Elements such as variable naming, assertion logic, and structural formatting were frequently described as "indistinguishable from human-written tests."

In 68% of cases, reviewers rated the LLM output as requiring only *minor edits* before being suitable for production use. On average, generating a complete script took under 30 seconds using an LLM compared to 5 to 15 minutes for equivalent manual implementation. Although human-authored scripts continued to edge out LLM outputs in terms of completeness and edge-case handling, the performance gap was not dramatic, especially for well-structured inputs.

These findings support the view that LLMs, while not yet a replacement for human testing expertise, can serve as highly effective accelerators in test development workflows particularly when used in conjunction with validation layers and human oversight.

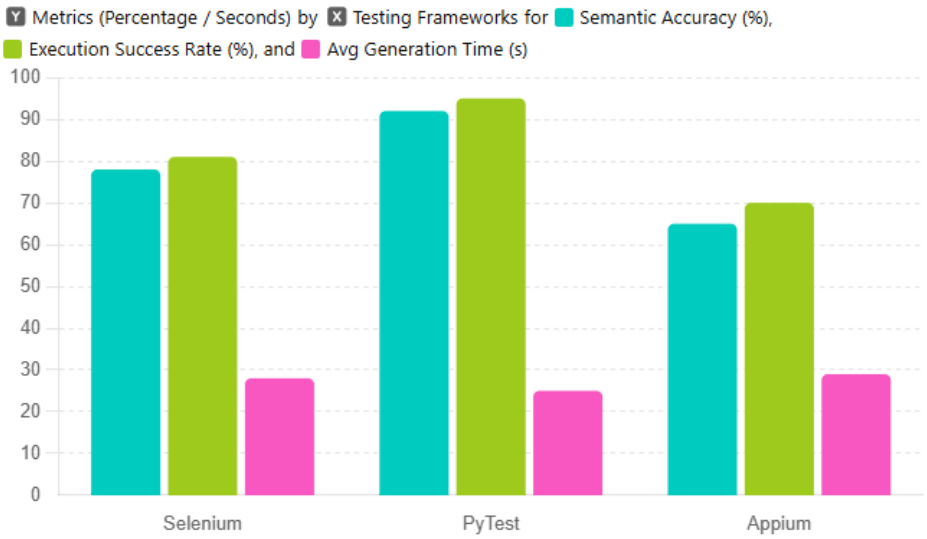

Diagram 2. Performance Comparison Across Testing Frameworks

Diagram 2 presents a comparative analysis of LLM performance across three widely used testing frameworks—Selenium, PyTest, and Appium—based on three key evaluation metrics: semantic accuracy, execution success rate, and average generation time. Displayed as a grouped bar chart, the figure highlights that PyTest achieved the highest overall scores,

with semantic accuracy exceeding 90% and minimal runtime failures, likely due to its declarative syntax and simpler execution environment. Selenium showed moderate performance, with slightly lower semantic alignment due to the complexity of UI interactions, while Appium trailed with the lowest scores, reflecting the challenges LLMs face in generating mobile test scripts that depend on dynamic UI elements and device-specific contexts. Generation time remained under 30 seconds for all frameworks, with negligible variance. This comparison underscores the strengths and limitations of current LLMs across different testing domains and emphasizes the need for tailored prompting and validation strategies per framework.

7. Discussion and Implications

7.1 Interpretation of Results

The findings presented in this study suggest that Large Language Models most notably GPT-4 are capable of producing executable software test scripts that are not only syntactically correct but also aligned with the semantic intent of natural language requirements. Although human-authored tests still exhibit an advantage in terms of handling edge cases and anticipating non-obvious behaviors, the gap is narrower than expected. The LLM-generated scripts performed reliably across multiple frameworks and application types, supporting our central hypothesis that test generation from natural language input is feasible and effective using current-generation models.

It is worth noting, however, that this performance is contingent on the structure and clarity of the input prompt. The better the contextual grounding and formatting of the requirement, the more likely the model is to produce meaningful output. This places some of the burden not on the model, but on the design of prompts and supporting input materials. Still, the overall results underscore the growing maturity of generative models in tackling practical software engineering tasks.

7.2 Practical Implications for Software Engineering

These findings have several concrete implications for software development practice. The potential to convert user stories or requirement documents directly into working test scripts opens up new possibilities for accelerating quality assurance processes particularly in agile settings where test maintenance often lags behind rapid code changes. Integrating LLMs into IDEs, test management systems, or CI/CD pipelines could help reduce manual scripting workloads, close documentation gaps, and promote more consistent test coverage.

For teams working in low-code or no-code environments, LLMs present an opportunity to lower the barrier to automation. Individuals without deep programming expertise could, in principle, describe intended behaviors in plain language and obtain structured, executable tests in return. Similarly, in projects involving legacy systems where formal documentation is often sparse LLMs may be useful in reverse-engineering test logic from inferred behavior or loosely defined specifications.

In these contexts, LLMs function less as replacements for human testers and more as accelerators: tools that can handle routine or boilerplate cases, freeing up developers to focus on more complex, high-risk, or exploratory testing tasks.

7.3 Limitations and Risks

Despite these strengths, the application of LLMs to test script generation carries several limitations that must be acknowledged. First and foremost, LLMs remain probabilistic systems. While they often generate plausible outputs, they do so without true understanding, and even small ambiguities in the prompt can result in incorrect or misleading code. We observed several cases of "logical hallucination" scripts that were structurally correct but failed to match the requirement’s intent, particularly when the specification lacked detail.

Additionally, successful execution is tightly coupled to the correctness of the runtime environment. Scripts may fail due to misconfigured environments, missing dependencies, deprecated APIs, or unstable selectors in UI testing contexts. These issues are external to the model but nevertheless affect perceived performance and reliability.

Finally, none of the evaluated models were fine-tuned specifically for the task of test generation. While general-purpose LLMs proved capable, there remains untapped potential in training models on domain-specific data albeit with the tradeoff of increased complexity, maintenance costs, and the risk of overfitting to narrow testing patterns.

7.4 Ethical and Organizational Considerations

Beyond the technical and practical aspects, the broader deployment of LLM-generated code including test scripts raises important ethical and organizational concerns. In high-assurance domains such as healthcare, aviation, or finance, reliance on machine-generated artifacts without sufficient human oversight could lead to silent test failures or undetected regressions. These risks are particularly acute in systems where test coverage is safety-critical and where any missed behavior may have real-world consequences. To address this, organizations adopting LLM-assisted testing must establish clear governance policies. Generated scripts should undergo the same validation, review, and traceability processes applied to human-written code. Moreover, audit trails must be preserved especially when test logic is derived from requirements that may evolve over time.

8. Conclusion and Future Work

8.1 Conclusion

This study set out to investigate the capabilities of Large Language Models (LLMs) in automating the generation of software test scripts from natural language requirements. Drawing on systematic experiments involving GPT-4, Codex, and Code Llama, we evaluated LLM performance across several metrics syntactic validity, semantic alignment, execution success, and test coverage using both Selenium and PyTest as representative frameworks. The results indicate that current-generation LLMs can reliably produce executable test scripts that align closely with the intent of the input specifications, requiring minimal post-editing in many cases.

While human-written tests continue to show advantages in nuanced areas such as edge-case reasoning and domain-specific interpretation, the gap is narrowing. For many practical use cases especially in agile, fast-paced, or resource-constrained environments LLMs offer a compelling alternative to manual scripting. Their ability to transform user stories or acceptance criteria into runnable test artifacts in seconds opens the door to new modes of collaboration between engineers, testers, and AI systems. Taken together, our findings affirm the feasibility of integrating LLMs into contemporary software testing workflows, not as replacements for human expertise, but as accelerators that complement and extend it.

8.2 Future Work

Several promising directions remain open for further exploration. One avenue involves the targeted fine-tuning of LLMs on domain-specific testing corpora, potentially combined with reinforcement learning from human feedback (RLHF) to refine the model's understanding of testing conventions and context-specific nuances. Such approaches may improve reliability in more complex or specialized scenarios.

In parallel, there is a need to broaden the scope of test types supported. Expanding beyond functional testing to include performance, security, and integration workflows would provide a more complete view of how LLMs might assist across the full spectrum of quality assurance. Incorporating structured requirement formats such as UML activity diagrams, state machines, or decision tables may also help mitigate prompt ambiguity and enhance the precision of generated outputs. Another critical area is post-generation validation. Automated tools that can verify API usage, detect flaky test patterns, and suggest refinements would greatly increase the robustness of LLM-generated scripts. This is particularly important as these tools move closer to integration with CI/CD systems, where false positives or brittle test logic can introduce costly friction. Future research should consider multilingual requirements and non-English development contexts, which remain underexplored. As enterprise adoption grows, there will also be increasing demand for privacy-preserving deployments of LLMs such as on-premise inference or federated architectures that respect data governance constraints while maintaining performance.

References

1) Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability, 22(5), 297–312.
2) Cadar, C., & Sen, K. (2013). Symbolic execution for software testing: Three decades later. Communications of the ACM, 56(2), 82–90.
3) Harman, M., & Jones, B. F. (2001). Search-based software engineering. Information and Software Technology, 43(14), 833–839.
4) Lucassen, G., Dalpiaz, F., van der Werf, J. M. E. M., & Brinkkemper, S. (2016). Improving agile requirements: the Quality User Story framework and tool. Requirements Engineering, 21(3), 383–403.

5) Fornaia, A. Midolo, G. Pappalardo and E. Tramontana, "Automatic Generation of Effective Unit Tests based on Code Behaviour," 2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Bayonne, France, 2020, pp. 213-218.
6) Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2016). DeepCoder: Learning to Write Programs. ArXiv, abs/1611.01989.
7) Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. W. (2020). A transformer-based approach for source code summarization. Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 4998–5007.
8) Yin, P., & Neubig, G. (2017). A syntactic neural model for general-purpose code generation. Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, 440–450.
9) Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), 1536–1547.
10) Panthaplackel, S., Li, J.J., Gligorić, M., & Mooney, R.J. (2020). Deep Just-In-Time Inconsistency Detection Between Comments and Source Code. AAAI Conference on Artificial Intelligence.
11) Tufano, M., Watson, C., Bavota, G., Poshyvanyk, D., & Di Penta, M. (2020). Unit test case generation with transformers and neural language models. In *Proceedings of FSE 2020*, 1110–1121.