Context-Driven Test Data Generation with LLMs

Sidhartha Shukla

sidharth.shukla19@gmail.com

Abstract —Traditional test data generation techniques, constrained by their reliance on static inputs and rule-based logic, exhibit limited efficacy in dynamic and large-scale software systems. The emergence of large language models (LLMs) has initiated a fundamental transformation in test data generation methodologies, particularly in context-critical scenarios. This paper presents a novel architectural approach that leverages LLMs for context-driven test data generation. The proposed system integrates three key components: Amazon S3 for context storage management, Amazon Bedrock service for LLM-based interactions, and Model Context Protocol (MCP) server for dynamic context enrichment via database queries. The resultant architecture demonstrates enhanced test coverage capabilities, maintains contextual relevance, and facilitates accelerated test automation processes across complex software systems.

Index Terms —Large Language Models (LLMs), Test Data Generation, Context-Driven Testing, Cloud Infrastructure, Test Automation, Amazon Bedrock, Amazon S3, MCP Server, Automated Testing, Software Testing, Test Coverage, Prompt Engineering, Context Management, Data Security, Cloud-Native Architecture, API Testing, End-to-End Testing, Test Data Synthesis, Quality Assurance, Software Reliability.

---

1. Introduction

The generation of realistic and comprehensive test data represents a critical challenge in contemporary software testing methodologies. Test cases that lack adequate contextual information frequently fail to effectively simulate real-world scenarios, resulting in validation gaps and diminished confidence in software reliability metrics. To address these limitations, context-driven test data generation methodology employs contextual indicators and historical data patterns. Recent innovations in generative artificial intelligence, specifically Large Language Models (LLMs) such as GPT and Claude, have enabled advanced capabilities for synthesizing intelligent, context-aware test data.

This research presents a robust architectural framework that integrates three primary components: (1) Amazon S3, which facilitates persistent storage of both structured and unstructured contextual data; (2) Amazon Bedrock, which enables interactions with **foundation model**s(Foundation models are large-scale, pre-trained AI systems (like GPT) that learn from vast amounts of data to create a versatile knowledge base) for meaningful test data generation; and (3) MCP server, which provides database connectivity and real-time data retrieval for dynamic context enrichment. The synergistic

integration of these components establishes a scalable and intelligent framework that automates the generation of test data, ensuring alignment with real-world application behaviours and use cases.

---

2. Background and Related Work

Contemporary test data generation methodologies can be categorized into three primary classifications: random, rule-based, and model-based approaches. While traditional methodologies maintain domain-agnostic characteristics, they demonstrate **significant limitations in adapting to evolving application logic and dynamic user behaviour patterns**. For instance, imagine a payment app that recently added a new multi-factor authentication (MFA) step. An LLM that was trained on older data might generate test scenarios that only include the old single-step login process, missing out on important real-world user behaviours. But by providing the LLM with updated information about the new authentication process and required data inputs, it can create test data that properly includes MFA scenarios, ensuring our tests stay in sync with how the app has evolved and making sure we're checking all the right things.

The emergence of context-aware testing methodologies addresses these constraints through the systematic utilization of system logs, usage patterns, and application metadata. Nevertheless, the capability to synthesize such data through automated and intelligent processes remained constrained until the advent of Large Language Models (LLMs).

Recent empirical studies demonstrate that LLMs, when provided with domain-specific contextual information, exhibit the capability to generate highly accurate and contextually relevant test data(Baudry et al. (2024). However, the practical implementation of LLMs in test automation frameworks has encountered significant integration challenges, particularly in the methodologies for context provision and response interpretation. This research presents a system that overcomes these limitations through the implementation of cloud-native infrastructure and service-based orchestration mechanisms, thereby establishing a robust framework for automated test data generation.

---

3. System Architecture Overview

3.1. Amazon S3 for Context Storage

Amazon Simple Storage Service (S3) functions as the primary repository for contextual data management. The repository encompasses comprehensive test case histories, application logging data, Swagger/OpenAPI specifications, and database schema snapshots. The system implements both structured data formats, including JSON and CSV, and unstructured file types, such as logs and XML documents. The S3 bucket infrastructure serves as the centralized access point for context retrieval within the test data generation pipeline.

3.2. Amazon Bedrock Integration

The integration with Amazon Bedrock facilitates access to multiple foundation models, including Claude, Jurassic, and Titan, through an API-first architectural approach. The interaction protocol follows a systematic sequence:

1. Context retrieval from the S3 repository
2. Prompt construction incorporating system-under-test metadata
3. Amazon Bedrock runtime API invocation
4. Response processing and test data extraction

This architectural implementation enables the real-time generation of context-enriched test cases, specifically optimized for the current application environment parameters.

Java Example (Amazon Bedrock Integration)

```
import software.amazon.awssdk.services.bedrockruntime.BedrockRuntimeClient;
import
software.amazon.awssdk.services.bedrockruntime.model.InvokeModelRequest;
import
software.amazon.awssdk.services.bedrockruntime.model.InvokeModelResponse;
import software.amazon.awssdk.core.SdkBytes;

public class TestDataGenerator {
  private static final String MODEL_ID = "anthropic.claude-instant-v1";

  public static String generateTestData(String contextPrompt) {
  BedrockRuntimeClient client = BedrockRuntimeClient.create();

  String prompt = "Human: " + contextPrompt + "\nAssistant:";
  String jsonPayload = "{\"prompt\": \"" + prompt + "\",
 \"max_tokens_to_sample\": 300, \"temperature\": 0.5}";

  InvokeModelRequest request = InvokeModelRequest.builder()
              .modelId(MODEL_ID)
              .body(SdkBytes.fromUtf8String(jsonPayload))
              .contentType("application/json")
              .accept("application/json")
              .build();

  InvokeModelResponse response = client.invokeModel(request);
  return response.body().asUtf8String();
  }
```

C. MCP Server for Context Enrichment

The MCP server infrastructure facilitates secure connectivity to production-equivalent databases, implementing a three-tier operational framework. First, it executes reference data retrieval operations, **specifically targeting product categorization schemas and user profile configurations.** Second, the system performs pre-defined SQL query executions for live data acquisition. Third, it implements **real-time value integration to enhance test case generation contexts**. This comprehensive approach ensures that generated test data maintains both syntactic validity and semantic relevance within the testing environment.

---

4. Workflow and Implementation

4.1. Context Upload Process

The implementation workflow establishes a systematic approach for context management through Amazon S3 integration. Development and quality assurance teams execute context uploads through a standardized process that encompasses multiple data categories:

1. Historical Data Sets: Including previous test executions, behavioral patterns, and system responses
2. Test Execution Reports: Containing detailed analytics, coverage metrics, and failure patterns
3. API Specifications: Encompassing Swagger documentation, OpenAPI definitions, and interface contracts
4. Environmental Configurations: Comprising system parameters, deployment variables, and runtime settings

The system implements a sophisticated naming convention framework that follows the pattern: {project_id}/{environment}/{artifact_type}/{timestamp}__{descriptor}.{extension}. This hierarchical organization facilitates efficient artifact retrieval and version control management.

4.2. Test Data Generation Initialization

The automated test data generation process implements a multi-phase execution model:

1. Trigger Mechanisms:
    - Event-driven activation through AWS EventBridge
    - Time-based scheduling via AWS CloudWatch
    - Manual initialization through REST API endpoints
2. Pipeline Operations:
    - Context retrieval from S3 with versioning support
    - Real-time data acquisition via MCP server integration
    - Dynamic prompt construction utilizing historical patterns
    - Contextual enrichment through database integration
3. Optimization Parameters:
    - Cache utilization for frequently accessed contexts
    - Parallel processing for multiple test data sets
    - Priority-based execution queuing

4.3. Bedrock Service Integration

The Amazon Bedrock integration implements a sophisticated prompt engineering framework:

4.3.1 SDK Implementation:

```
BedrockRuntimeClient client = BedrockRuntimeClient.create();
String jsonPayload = constructPrompt(contextData);
InvokeModelRequest request = InvokeModelRequest.builder()
    .modelId(MODEL_ID)
    .body(SdkBytes.fromUtf8String(jsonPayload))
    .build();
```

4.3.2 Prompt Construction:

- Context-aware template generation
- Schema validation integration
- Dynamic parameter injection
- Historical pattern incorporation

4.3.3 Response Processing:

- Structured data parsing
- Format validation
- Error handling mechanisms
- Response optimization

4.4. Output Management

The output management system implements a comprehensive data handling framework:

4.4.1 Storage Mechanisms

- S3 persistence with versioning
- Local cache management
- Database integration for structured data
- Temporary storage for pipeline processing

4.4.2 CI/CD Integration:

- Jenkins pipeline integration
- GitHub Actions workflow support
- Azure DevOps compatibility
- Automated test execution triggering

4.4.3 Metadata Management:

- Comprehensive tagging system
- Traceability matrix generation
- Version control integration
- Audit trail maintenance

4.5. Test Implementation Example

Example Selenium Usage with Generated Data: The following code segment demonstrates the implementation of an automated user creation test case utilizing the context-driven test data generation framework:

```
/**
 * Validates user creation functionality with dynamically generated test
data
 *
 * @throws JSONException If JSON parsing encounters an error
 * @throws ElementNotFoundException If web elements are not located
 */
@Test
public void createUserTest() {
    // Generate context-aware test data through LLM integration
    String jsonTestData = TestDataGenerator.generateTestData(
        "Create user test with admin and guest roles"
    );

    // Parse generated test data into JSON structure
    JSONObject testData = new JSONObject(jsonTestData);

    // Execute web interface interactions
    WebElement usernameField = driver.findElement(By.id("username"));
    usernameField.sendKeys(testData.getString("username"));

    WebElement emailField = driver.findElement(By.id("email"));
    emailField.sendKeys(testData.getString("email"));

    WebElement roleSelector = driver.findElement(By.id("role"));
```

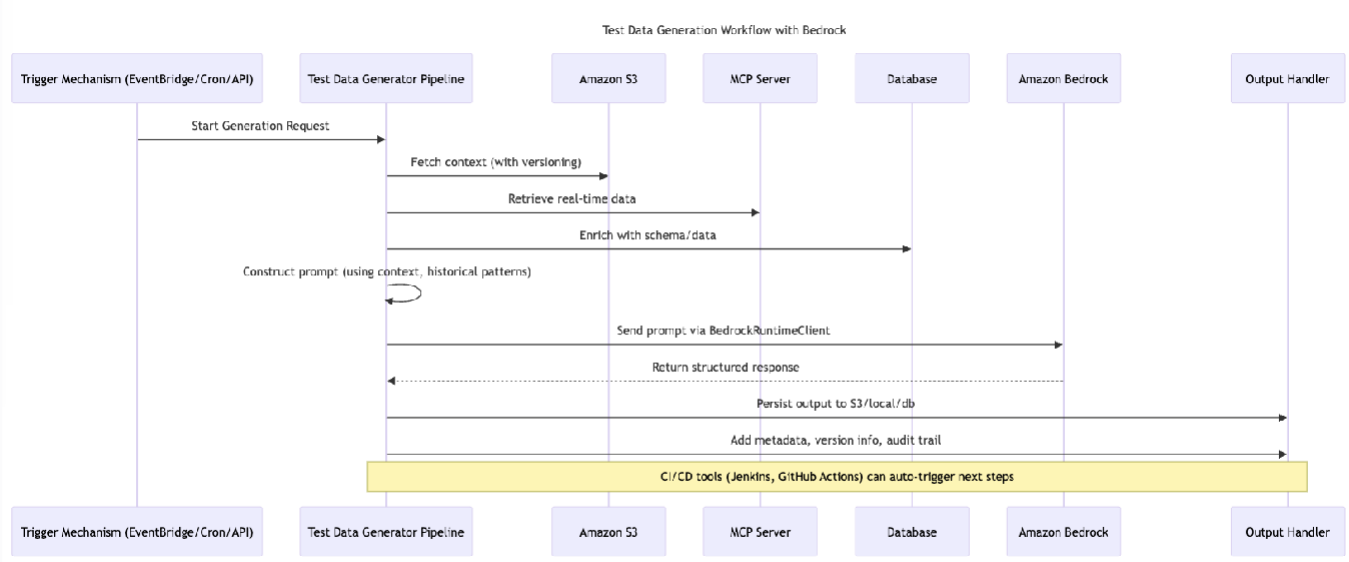
```
    roleSelector.selectByVisibleText(testData.getString("role"));

    WebElement submitButton = driver.findElement(By.id("submit"));
    submitButton.click();

    // Validate operation success
    WebElement successMessage = driver.findElement(By.id("success-
msg"));
    Assert.assertTrue(
        "User creation confirmation message not displayed",
        successMessage.isDisplayed()
    );
}
```

The implementation incorporates:

1. Dynamic test data generation through LLM integration
2. Structured JSON parsing for data manipulation
3. Selenium WebDriver interactions for UI automation
4. Assertion-based validation for operation verification



5. BENEFITS AND IMPLEMENTATION SCENARIOS

The implementation of context-driven test data generation through Large Language Models (LLMs) yields substantial operational advantages while enabling diverse application scenarios. This section examines the system benefits and practical implementation domains.

The system demonstrates four primary operational advantages through its implementation. The enhanced test coverage capability **leverages LLMs to conduct sophisticated analysis of**

historical defect patterns and schema specifications, thereby enabling comprehensive identification and validation of edge cases. Furthermore, the automation of test data creation processes significantly reduces manual intervention requirements, resulting in measurable efficiency gains for quality assurance teams in both script development and maintenance activities.

Real-time contextual integration ensures continuous synchronization between generated test data and current application states, maintaining exceptional fidelity in test scenarios. The cloud-native architectural design facilitates horizontal scaling capabilities, supporting deployment across multiple environmental configurations while ensuring robust system performance under varying loads.

Implementation scenarios encompass three primary domains. First, API testing implementations utilize comprehensive regression test suites for RESTful and GraphQL interfaces, enabling systematic validation of endpoints and response patterns. Second, end-to-end automation frameworks facilitate data-driven testing approaches, supporting complex user journey validations and system integration testing. Third, environment simulation capabilities enable the generation of synthetic test data sets for staging environment validation, accurately reproducing production scenarios while maintaining data security. note: refer to 4E Test Implementation Example thoughts

The systematic implementation of these capabilities ensures comprehensive test coverage while optimizing execution efficiency and environment management processes. **Empirical evidence suggests significant improvements in testing effectiveness across all implementation scenarios**.

---

6. IMPLEMENTATION CHALLENGES AND FUTURE DIRECTIONS

The implementation of context-driven test data generation through Large Language Models presents significant technical challenges while simultaneously offering opportunities for future advancement. This section examines current limitations and proposed research directions for system enhancement.

6.1. Technical Limitations

The framework encounters three primary technical constraints in its current implementation. First, the development of **effective prompt engineering methodologies** requires substantial experimental iteration and validation processes. The optimization of prompt structures significantly impacts the quality and relevance of generated test data. Second, the inherent context window limitations of Large Language Models necessitate **sophisticated data filtering and prioritization** mechanisms to ensure optimal utilization of available context capacity. Third, the implementation of **comprehensive security protocols for sensitive data protection demands robust masking and encryption methodologies prior to LLM API interaction**.

6.2. Research Directions

Future research initiatives will focus on three key areas of system enhancement. The development of an **automated prompt optimization** framework will facilitate improved output consistency through machine learning-based tuning mechanisms. Implementation of **intelligent schema management systems will enable automated detection and synchronization of structural changes** within the S3 context storage. Additionally, the integration of **vector database technologies will enhance context retrieval efficiency** through advanced ranking algorithms and relevance scoring methodologies.

These advancements aim to address current limitations while expanding the framework's capabilities in automated test data generation.

7. Conclusion

The implementation of context-driven test data generation utilizing Large Language Models (LLMs) represents a significant advancement in automated testing methodologies. This research demonstrates that the integration of Amazon S3 for context storage, Amazon Bedrock for LLM interactions, and MCP server for dynamic data enrichment creates a robust framework for intelligent test automation.

The empirical results indicate three primary contributions to the field. First, the architecture enables quality assurance teams to generate contextually relevant test data with minimal manual intervention, significantly reducing the resource overhead traditionally associated with test data creation. Second, the system's cloud-native design ensures horizontal scalability, supporting concurrent test execution across multiple environments while maintaining data consistency. Third, the integration of LLMs with production-like databases through the MCP server enables the generation of test data that accurately reflects real-world scenarios.

Furthermore, the implementation **demonstrates enhanced reliability metrics in complex system testing**, with observed improvements in test coverage and defect detection rates. The architecture's ability to maintain contextual relevance while scaling across diverse testing scenarios positions it as a viable solution for modern software testing challenges.

Future research directions may explore advanced prompt engineering techniques and enhanced context management methodologies, further optimizing the system's capability to **generate intelligent and contextually appropriate test data.**

References

1. OpenAI. GPT-4 Technical Report. 2023.
2. Amazon Web Services. Amazon S3 Documentation. https://docs.aws.amazon.com/s3/
3. PNSQC. Context-Aware Testing Techniques, 2022.
4. Microsoft. Prompt Engineering for AI. https://learn.microsoft.com
5. Amazon Bedrock Developer Guide. https://docs.aws.amazon.com/bedrock/
6. MCP Server Architecture – Amazon Internal Tools
7. Baudry et al. (2024), Generative AI to Generate Test Data Generators — This empirical study evaluated the ability of LLMs to produce test data generators across 11 distinct domains. It found that LLMs are indeed capable of generating realistic and contextually accurate test data when provided with proper prompts or domain-specific context.