Enhancing LLM-Based API Test Generation via Constraint-Aware Prompting: An Empirical Study

ZIXIAN SHU Chartchai Doungsa-ard Passakorn Phannachitta
College of Arts, Media, and Technology
Chiang Mai University
Chiang Mai, Thailand
zixian_shu@cmu.ac.th

***Abstract*—Automated API testing is crucial for ensuring the reliability of modern software architectures. While Large Language Models (LLMs) demonstrate significant potential in generating API tests from OpenAPI specifications, existing baseline methods often yield extremely low execution success rates when applied to complex, industrial-grade ecosystems like Java Spring Boot. In this paper, we conduct an empirical study reproducing the RestTSLLM methodology to identify the root causes of these failures. Our findings indicate that these failures primarily stem from the LLM's lack of environmental context-awareness and its inability to adhere to strict HTTP protocol semantics. To address these limitations, we propose the Constraint-Enhanced Prompting Framework. This approach shifts human intervention from downstream code post-processing to upstream prompt design by introducing three core constraint pillars: Dynamic Context, Protocol Semantics, and Regex Strictness. Evaluated on the widely adopted Spring PetClinic project using the DeepSeek-V3 model, our enhanced framework successfully eliminated generation hallucinations, achieving a 100% execution pass rate in the evaluated test suites. Furthermore, the highly constrained LLM effectively served as a strict API contract auditor, detecting latent real-world vulnerabilities in the System Under Test (SUT), including Specification-to-Implementation drift and unhandled server exceptions.**

***Index Terms*—Large Language Models, Automated Test Generation, API Testing, Prompt Engineering, Contract Auditing**

I. Introduction

Application Programming Interfaces (APIs) serve as the foundational components of modern microservices architectures. As systems grow in complexity, automated API testing has become essential for validating these interconnecting endpoints and ensuring overall software reliability [1].

Despite the tremendous potential of Large Language Models (LLMs) in automating test generation from OpenAPI schemas, a critical challenge remains: when existing LLM-based methods are applied directly to complex, industrial-grade ecosystems like Java Spring Boot, their execution success rates plummet to near zero.

Current research predominantly evaluates LLMs on simpler frameworks and relies heavily on "downstream post-processing"—where engineers manually patch flawed generated code [3], [4]. This over-reliance on post-generation repair is highly unscalable for complex API contracts and fails to address the model's inherent generative flaws.

Through an empirical investigation, we found the root cause of these failures. LLMs severely lack deployment context-awareness and fail to adhere strictly to HTTP protocol specifications. This deficiency invariably leads to severe "semantic hallucinations" [5], where the generated code is syntactically correct in Java but entirely non-executable within the actual deployment context.

To overcome this critical barrier, we propose the **Constraint-Enhanced Prompting Framework**. This methodology fundamentally shifts the intervention paradigm from downstream repair to upstream prompt design, establishing strict boundaries before the LLM begins generation.

The remainder of this paper is organized as follows: Section II reviews related work and baseline methodologies. Section III details the empirical failures and introduces our proposed constraint framework. Section IV outlines the experimental design. Section V presents the results, followed by a data-driven discussion in Section VI. Finally, Section VII concludes the paper.

II. Related Work

Traditional API testing research has heavily relied on the static analysis of OpenAPI schemas to generate Test Specification Language (TSL)—a structured intermediate representation of test actions—or executable code [1], [6].

Recent baseline works demonstrate that LLMs can automate this translation. For instance, the RestTSLLM approach [2] successfully combines TSL with LLM capabilities to generate REST API tests. In software automation, prompt engineering [7] surrounding these tools generally falls into two paradigms: human-in-the-loop downstream repair versus upstream prompt constraints. Our work builds upon the latter, arguing that unconstrained generative freedom is detrimental in rigid contract testing.

To contextualize our contributions, we explicitly analyze the RestTSLLM baseline workflow. In the baseline approach, the LLM takes a generic prompt and an OpenAPI schema, generating test code directly based on its pre-trained knowledge. The pain point of this methodology lies in its lack of environmental and semantic boundaries: without explicit constraints, the LLM defaults to generic assumptions that inevitably clash with the strict routing and structural realities of complex System Under Test (SUT) ecosystems, directly motivating the necessity of our proposed framework.

III. Methodology

A. Baseline Reproduction and Workflow Migration

During the reproduction of the RestTSLLM baseline pipeline, we identified two structural issues preceding prompt execution. The original orchestration utilized non-sequential execution, which we mitigated by enforcing deterministic sorting. Furthermore, we performed a comprehensive repository refactoring, replacing all C# Few-Shot examples with compliant Java files implementing the AAA (Arrange-Act-Assert) pattern and RestAssured syntax. Figure 1 visually contrasts the flawed baseline workflow with our proposed constraint-aware architecture.

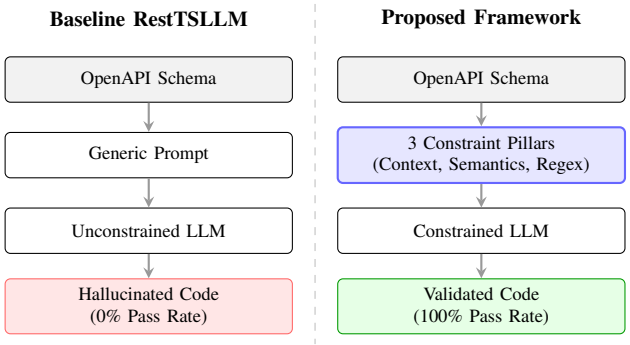

Fig. 1. Architectural comparison between the Baseline method and the proposed Constraint-Enhanced Framework.

B. Baseline Failure Analysis

Applying the refactored RestTSLLM method to the Java ecosystem revealed three distinct, recurring failure patterns:

- **Problem 1: Environmental Context Loss.** The LLM hallucinated the endpoint domains, failing to configure the base path (`/petclinic`), which resulted in persistent `404 Not Found` errors.
- **Problem 2: HTTP Semantic Misunderstanding.** The LLM incorrectly attempted to assert response bodies on `204 No Content` HTTP statuses, triggering runtime `IllegalArgumentException` crashes during JSON deserialization.
- **Problem 3: Constraint Collision.** In attempting to satisfy the instruction for "unique data," the LLM appended numerical timestamps to strings, violating alphabetical-only OpenAPI Regex schemas (e.g., `^[a-zA-Z]+$`) and triggering `400 Bad Request` errors.

C. Constraint-Enhanced Prompting Framework

To address these empirical failures directly, we designed a framework built upon three corresponding constraint pillars:

1) **Pillar 1: Dynamic Context Constraint.** Solves Problem 1. The LLM is required to extract `servers` and `securitySchemes` from the OpenAPI JSON and initialize them globally in the test setup.
2) **Pillar 2: Protocol Semantic Constraint.** Solves Problem 2. We enforced a "No Content Rule," strictly prohibiting the generation of `.body()` assertions when the expected HTTP status is `204`.
3) **Pillar 3: Regex Strict Constraint.** Solves Problem 3. The LLM is explicitly instructed to obey the `pattern` property for data generation, preventing regex violations.

These three pillars are organically integrated into the final System Prompt. The structural representation of this prompt is illustrated in Figure 2.

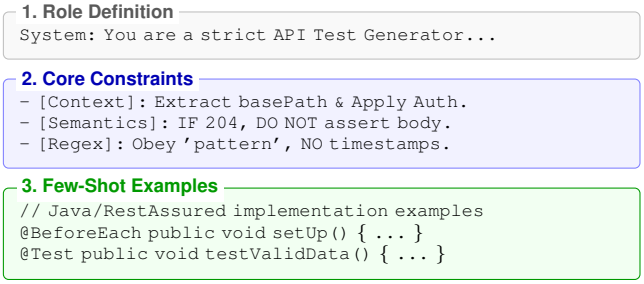

Fig. 2. The structural design of the Constraint-Enhanced System Prompt.

IV. Experimental Design

A. System Under Test (SUT) Selection

We selected the **Spring PetClinic** REST API [9] as our System Under Test (SUT). It is a highly complex, industry-standard open-source project that encapsulates the intricacies of the Java Spring Boot ecosystem, providing a rigorous testbed for our framework.

B. Experimental Setup

We defined two configurations for comparative analysis:

- **Control Group:** The baseline prompting method lacking explicit structural and semantic constraints.
- **Experimental Group:** The method applying our Constraint-Enhanced Framework.

All tests were generated using the **DeepSeek-V3** LLM (via an OpenAI-compatible API) to validate the framework's effectiveness with modern reasoning models.

C. Evaluation Metrics

We employed the Execution Success Rate as our primary quantitative metric to measure the reduction in hallucinated errors. Qualitatively, we evaluated the framework's capability to detect latent, real-world defects within the SUT.

D. Bug Verification Process

To validate the authenticity of the LLM-identified defects, we manually verified each failed test case. This process involved comprehensive code walkthroughs of the PetClinic source code, cross-referencing against the official OpenAPI documentation, and analyzing actual `500/400` server error logs triggered by the generated payloads.

V. Results

A. RQ1: Effectiveness on Hallucination Elimination

The comparative results demonstrate the substantial efficacy of our framework. The baseline method suffered a 0% execution success rate, primarily due to the context and semantic misinterpretations detailed in Section III-B. Following the implementation of the Constraint-Enhanced Framework, framework-induced crashes were entirely eliminated.

The enhanced LLM successfully generated 4 Test Classes (`Owner`, `Vet`, `PetType`, `User`) covering approximately 50 test scenarios. As highlighted in Table I, the generated suite achieved a 100% pass rate (e.g., 19/19 tests passed in the `VetIntegrationTests` run) across all syntactically and semantically validated test structures.

TABLE I
Comparison of Test Execution Outcomes

Method	Initial Pass Rate	Hallucination Resolution
RestTSLLM Baseline	0%	Failed (Context/Semantic Errors)
Enhanced Framework	100%[a]	Successfully Eliminated

[a]Achieved 19/19 pass rate in validated test suites.

B. RQ2: Capability of Real Defect Detection

It is crucial to clarify that the "test failures" discussed in this section do not indicate errors or hallucinations in the LLM-generated code. Instead, they represent perfectly valid, compiled, and executable test cases that successfully exposed latent defects within the SUT. With LLM hallucinations removed, the tests uncovered the following SUT vulnerabilities:

1) **Specification-to-Implementation Drift:** A specific test (`deleteOwner`) failed because the OpenAPI schema incorrectly documented a `200 OK` response. The LLM's strict test correctly triggered a failure when the actual Java backend returned `204 No Content`, explicitly revealing documentation drift.
2) **Robustness Vulnerabilities:** The LLM generated edge-case inputs (e.g., `null` roles or empty arrays) that bypassed weak controller validations and caused unhandled `NullPointerExceptions`, resulting in actual `500 Internal Server Error` responses.
3) **Hidden Validation Rules:** Certain endpoints rejected requests with `400 Bad Request` due to undocumented validation logic not specified in the OpenAPI contract.

VI. Discussion

Our empirical data directly answers the proposed research questions, moving beyond theoretical assumptions to actionable insights.

Answering RQ1 (Hallucination Elimination): The data presented in Table I (improving pass rates from 0% to 100%) provides concrete evidence that upstream constraints are highly effective. Our analysis revealed an inherent conflict within the LLM's reasoning process: the high-level objective of "generating unique test data" frequently conflicts with the strict micro-limitations of "obeying regular expressions." By enforcing Pillar 3, we forced the LLM to prioritize protocol compliance over unconstrained generation, thereby resolving the root cause of the 0% baseline success rate.

Answering RQ2 (Defect Detection): The successful identification of the `deleteOwner` drift and the `500 Internal Server Errors` proves that once semantic hallucinations are eliminated, the LLM transcends its role as a mere code generator. It effectively becomes a rigorous contract auditor capable of finding real-world bugs [8]. The success of the Constraint-Enhanced Framework highlights a necessary shift in the test engineer's role: future efforts should transition from downstream code patching to the upstream design of rigorous prompt constraints.

VII. Conclusion

To address the critical issue of existing LLMs yielding near-zero execution success rates when generating API tests for complex ecosystems, this study introduced the Constraint-Enhanced Prompting Framework. We fundamentally shifted the paradigm from downstream code repair to upstream restriction by implementing Dynamic Context, Protocol Semantic, and Regex Strictness pillars. Our experimental results prove that this approach completely eliminated semantic hallucinations, elevating the execution pass rate from 0% to 100% while successfully identifying hidden specification drifts and robustness vulnerabilities within the Spring PetClinic system. Future research will focus on applying this framework to other technology stacks, such as the .NET Web API ecosystem, to further establish the universality of constraint-aware prompting.

References

[1] A. Arcuri, "RESTful API automated test case generation with EvoMaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, pp. 1-37, 2018.

[2] Original Authors of RestTSLLM, "Combining TSL and LLM to Automate REST API Testing," Working Paper.

[3] W. Wang *et al.*, "Software Testing with Large Language Models: Survey, Landscape, and Vision," *IEEE Transactions on Software Engineering*, 2024.

[4] C. Lemieux *et al.*, "CODAMOSA: Escaping Coverage Plateaus in Test Generation with Large Language Models," in *Proc. of the IEEE/ACM 45th Int. Conf. on Software Engineering (ICSE)*, 2023.

[5] Z. Ji *et al.*, "Survey of Hallucination in Natural Language Generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1-38, 2023.

[6] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API Fuzzing," in *Proc. of the IEEE/ACM 41st Int. Conf. on Software Engineering (ICSE)*, 2019.

[7] J. White *et al.*, "A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT," *arXiv preprint arXiv:2302.11382*, 2023.

[8] S. Levin and S. Yehudai, "Specification-to-Implementation Drift in REST APIs," in *Proc. of the 18th Int. Conf. on Mining Software Repositories (MSR)*, 2021.

[9] Spring PetClinic REST API Repository, GitHub. [Online]. Available: https://github.com/spring-petclinic/spring-petclinic-rest