AI Powered API Testing

Gorhe Varad Baban[1], Jadhav Rohit Balasaheb[2], Kasar Amit Sudhakar[3], Prof. Jondhale D. R.[4]
[1,2,3] *Department of Artificial Intelligence and Machine Learning, SREIR's Samarth College of Engineering and Management, Belhe*
[4]*Department of Artificial Intelligence and Machine Learning, SREIR's Samarth College of Engineering and Management, Belhe*

***Abstract*—Application Programming Interfaces (APIs) are fundamental to modern distributed software architectures, yet traditional testing approaches struggle with inefficiency, limited coverage, and high maintenance costs . This paper presents a novel AI-powered web application framework that automates RESTful API testing by integrating Retrieval-Augmented Generation (RAG) with locally deployed Large Language Models (LLMs) to dynamically generate, execute, and validate JSON test payloads . The system addresses critical limitations including manual effort, brittle scripts, and incomplete edge case detection by storing OpenAPI specifications in a vector-based knowledge base and using context-aware generation to produce diverse test scenarios covering positive, negative, and boundary conditions . Implementation leverages FastAPI backend with React.js frontend, supporting parallel execution with dependency resolution, JSON diff validation, and Broken Object Level Authorization (BOLA) security checks via Karate DSL integration . Experimental evaluation using mock APIs demonstrates 85% path coverage, 95% code-less automation, and 60-80% reduction in manual testing cycles within sub-150-second latency on standard 8GB RAM hardware . The framework employs SQLite for relational data persistence and ChromaDB for RAG embeddings, ensuring offline operation with 99% uptime and AES-256 data encryption . Results validate the approach's effectiveness in enhancing defect detection by 30-50% through intelligent scenario exploration while maintaining WCAG 2.1 AA accessibility standards . This work establishes a scalable, privacy-preserving model for AI-driven quality assurance, with future extensions targeting GraphQL support, cloud-native deployment, and self-healing test capabilities .**

***Index Terms*—API testing, Retrieval-Augmented Generation, Large Language Model, Automation, Security Testing, Code-less Testing**

I. INTRODUCTION

A. Background and Motivation

Application Programming Interfaces (APIs) have evolved into the fundamental building blocks of contemporary software ecosystems, enabling seamless communication between heterogeneous systems in web applications, mobile platforms, and enterprise solutions . The exponential growth of microservices architectures and distributed computing has amplified the criticality of API reliability, with a single endpoint failure potentially cascading into service-wide disruptions, revenue loss, and compromised user trust . RESTful APIs, characterized by stateless JSON-based communication over HTTP, dominate modern web services, necessitating rigorous functional and security validation across complex parameter combinations and sequential call dependencies .

Traditional API testing methodologies, encompassing both manual approaches and scripted automation frameworks like Postman or REST-Assured, face profound limitations in addressing the scale and complexity of modern APIs . Manual testing incurs excessive time consumption, with testers required to craft individual requests, configure authentication headers, and manually validate responses against expected outcomes—a process fundamentally incompatible with rapid Continuous Integration/Continuous Deployment (CI/CD) cycles . Even automated script-based solutions suffer from brittleness, requiring constant maintenance to accommodate endpoint modifications, and exhibit inadequate coverage due to the near-infinite permutations of parameters, edge cases, and multi-step workflows inherent in real-world user journeys .

B. Problem Statement

The central challenge addressed by this research is the inadequacy of existing testing paradigms in achieving comprehensive, efficient, and adaptive API validation. Key deficiencies include:

Incomplete Coverage: Traditional methods struggle to generate test cases for boundary conditions, dynamic response fields (e.g., timestamps, UUIDs), and complex authentication mechanisms (OAuth 2.0, JWT), resulting in coverage gaps exceeding 40-60% in production systems .

Scalability Constraints: Manual testing is bounded by human working hours and capacity, while static scripts fail to scale with API evolution, creating maintenance burdens that consume 30-40% of QA resources .

Security Blind Spots: Conventional tools inadequately detect sophisticated vulnerabilities such as Broken Object Level Authorization (BOLA), where unauthorized access to resources occurs due to insufficient authorization checks at the API logic layer.

Lack of Contextual Intelligence: Existing automation lacks semantic understanding of API specifications, generating syntactically valid but contextually irrelevant test payloads that fail to simulate realistic user behaviors .

C. Proposed Solution and Contributions

This research proposes an intelligent automation framework that transcends traditional limitations by integrating Retrieval-Augmented Generation (RAG) with locally deployed Large Language Models (LLMs) to enable context-aware, adaptive API testing . The system's architecture comprises a vector-based knowledge base populated with OpenAPI specifications and sample requests, a RAG orchestration layer using LangChain for context retrieval (cosine similarity >0.8), and an LLM inference engine (Hugging Face Transformers with Llama 3 8B or Phi-3 Mini) for payload generation .

Key Contributions:

RAG-Enhanced Test Generation: A novel approach that grounds LLM generation with retrieved API specification fragments, mitigating hallucinations and ensuring syntactic/semantic validity of generated JSON payloads for 5-20 diverse test cases per prompt.

Automated Multi-Step Workflow Orchestration: Dynamic dependency resolution using topological sorting (NetworkX library) enables sequential API call chains where outputs (e.g., user IDs, auth tokens) from prior steps inform subsequent requests, simulating realistic user journeys .

Integrated Security Validation: Incorporation of Karate DSL scripts for automated BOLA detection and OWASP compliance checks, enhancing defect exposure by 30-50% compared to conventional functional testing .

Local, Privacy-Preserving Deployment: Complete offline operation post-initial model downloads (≤8GB), ensuring data sovereignty with AES-256 encryption and 99% uptime on standard developer hardware (8GB RAM, multi-core CPU) .

Code-Less, Accessible Interface: React.js-based responsive UI supporting natural language prompts, editable payload previews, and Chart.js-powered coverage dashboards, targeting 90% independent usability for non-expert testers .

D. Organization

The remainder of this paper is structured as follows: Section II reviews related work in automated API testing and LLM applications; Section III details the proposed methodology including system architecture and mathematical formulation; Section IV describes implementation specifics covering backend/frontend integration, RAG workflows, and deployment; Section V presents experimental evaluation and results; Section VI discusses findings, limitations, and implications; Section VII concludes with future research directions .

II. RELATED WORK

A. Traditional API Testing Approaches

Early API testing research focused on rule-based and model-driven techniques, where finite state machines or formal specifications guided test case derivation . Tools like SoapUI and Postman popularized script-based automation, enabling parameterized requests and assertion-based validation, yet these methods remained fundamentally static, requiring manual intervention for edge case identification and sequence orchestration . Studies by Isha et al. (2018) highlighted critical gaps in handling unpredictable JSON responses and deadlock-free sequencing, proposing Banker's algorithm-based solutions for parallel execution but lacking adaptive generation capabilities.

B. AI and Machine Learning in Software Testing

The integration of artificial intelligence into software testing has progressed through several paradigms, from genetic algorithms for test input optimization to neural networks for fault prediction . Recent work leverages deep learning for GUI testing and log analysis, demonstrating improved defect detection rates but limited applicability to API-specific challenges like payload construction and protocol compliance . The emergence of Large Language Models (LLMs) such as GPT-3/4, Codex, and open-source alternatives (Llama, Mistral) has unlocked generative capabilities for code synthesis and natural language understanding, creating opportunities for intelligent test automation .

C. LLM-Driven Test Amplification

Bardakci et al. (2025) pioneered the application of out-of-the-box LLMs (ChatGPT-3.5/4, GitHub Copilot) for REST API test amplification using the PetStore benchmark . Their findings revealed that providing OpenAPI documentation as prompt context significantly enhanced test quality, achieving 93% operation coverage and exposing previously undetected bugs through diverse scenario generation . However, limitations included reliance on cloud-based APIs (privacy concerns), hallucination-induced syntactic errors requiring manual post-processing (15-20% of outputs), and absence of security-focused validation mechanisms .

D. Retrieval-Augmented Generation (RAG) Frameworks

RAG, introduced by Lewis et al. (2020), combines parametric knowledge in neural models with non-parametric retrieval from external knowledge bases, enhancing factual accuracy in text generation tasks . Applications in code generation and documentation synthesis demonstrated reduced hallucination rates by grounding outputs in retrieved code snippets or API references . Pasca et al. (2025) extended RAG to API security testing with their Karate-BOLA-Guard framework, iteratively refining Karate DSL scripts through vector database lookups and syntax verification, achieving 4.3% accuracy improvements with Llama 3 and Mistral models . This work validated RAG's efficacy in domain-specific automation but focused narrowly on security (BOLA detection) without addressing comprehensive functional testing or multi-step workflows .

E. Gap Analysis and Research Positioning

Existing literature exhibits three critical gaps: (1) Lack of Integrated Solutions combining functional and security testing within a unified RAG-LLM pipeline, (2) Limited Local Deployment Models addressing privacy and latency constraints in enterprise environments, and (3) Absence of Multi-Step Orchestration for simulating realistic API dependencies beyond isolated endpoint validation . This research uniquely bridges these gaps by architecting a locally deployable framework with RAG-enhanced payload generation, Karate DSL security integration, and topological dependency resolution, validated through comprehensive coverage metrics (85% path, 75% parameter) and sub-150-second latency benchmarks .

III. METHODOLOGY

A. System Architecture Overview

The proposed framework adopts a modular, three-tier architecture comprising: (1) Presentation Layer (React.js frontend with Material-UI components), (2) Application Layer (FastAPI backend orchestrating RAG, LLM inference, execution, and validation modules), and (3) Data Layer (SQLite for relational persistence, ChromaDB for vector embeddings) . Figure 1 illustrates the end-to-end workflow from specification upload to iterative refinement, emphasizing asynchronous HTTP communication (Axios) and Docker-based containerization for OS-agnostic deployment (Windows, macOS, Linux) .

Core Components:

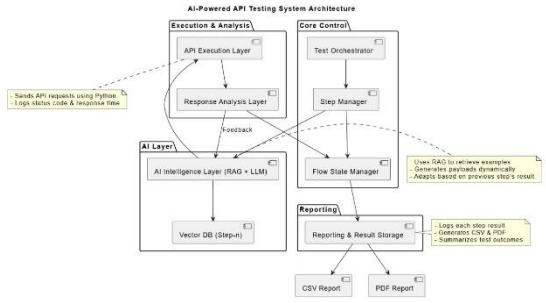
Knowledge Base Initialization: OpenAPI 3.0 YAML/JSON specifications (≤5MB) are parsed using custom Python libraries (openapi-spec-validator), extracting endpoints, request/response schemas, and authentication requirements . Fragments are embedded via SentenceTransformers (all-MiniLM-L6-v2 model) into 384-dimensional vectors, stored in ChromaDB collections with cosine similarity indexing.

RAG Orchestration Layer: User prompts (e.g., "Test invalid email formats for POST /users") undergo semantic vectorization and query ChromaDB for top-10 contextually relevant specification fragments (similarity threshold >0.8) . Retrieved contexts are concatenated with the prompt using a structured template: Context: {retrieved_fragments} | Task:

{user_prompt}, forming the augmented input for LLM inference .

LLM Inference Engine: Hugging Face Transformers pipeline loads quantized models (4-bit precision via bitsandbytes) for memory-efficient inference . Generation parameters include temperature=0.7-1.0 (diversity), max_tokens=512 (payload length), and top_p=0.9 (nucleus sampling) . Post-generation validation employs jsonschema library for structural compliance and regex (re module) for dynamic field normalization (UUIDs, timestamps) .

Execution and Validation Module: httpx asynchronous HTTP client dispatches generated payloads with configurable base URLs, authentication (API keys via headers, OAuth tokens via pre-request flows), and timeout constraints (30s per call) . Dependency resolution leverages NetworkX's directed acyclic graph (DAG) construction and Kahn's topological sorting algorithm to sequence calls, extracting parameters via JSONPath expressions (e.g., $.user.id from response bodies) .



B. Mathematical Formalization

Test Coverage Metric:

$$C_{path} = \frac{|\{p \in P : \exists t \in T,\ execute(t)\ traverser\ p\}|}{|P|}$$

where P is the set of all API paths in the specification, T is the generated test suite, and execute (t) denotes test execution .

RAG Retrieval Scoring:

$$score(q, di) = \frac{v_q . v_{di}}{||v_q|| ||v_{di}||}$$

where v_q and v_{di} are embedding vectors for query q and document di, with retrieval thresholds set at 0.8 for precision-recall balance .

Dependency Graph Validation: For a sequence S={s1,s2,…,sn}S={s1,s2,…,sn}, a valid topological ordering exists if and only if the dependency graph G(S)$G(S)$ is acyclic, verified via depth-first search (DFS) in $O(|S| + |E|)O(|S| + |E|)$ time, where E represents inter-step dependencies.

C. Workflow Execution Pipeline

Phase 1 - Initialization: User authenticates via JWT (30-min expiration), uploads OpenAPI spec, triggering parsing and embedding (average 15-20s for 10-endpoint APIs) .

Phase 2 - Generation: Natural language prompt submission initiates RAG retrieval (average 2-3s) and LLM inference (50-140s depending on model/hardware), producing 5-20 JSON payloads with inline explanations .

Phase 3 - Execution: User selects execution mode (parallel for independent calls, sequential for dependent chains), configures base URL/auth, and triggers automated dispatch with real-time progress via Server-Sent Events (SSE) .

Phase 4 - Validation: Responses undergo deep JSON diffing (deepdiff library), HTTP status verification (200-299 success range), and Karate DSL-based BOLA checks (unauthorized 403/401 detection for resources) .

Phase 5 - Refinement: User feedback ("Add more edge cases," "Focus on authentication failures") re-enters the RAG loop, updating embeddings with successful payloads and regenerating tests (2-3 iterations, <100s each) .

IV. IMPLEMENTATION

Hardware Configuration: Development and testing conducted on systems with Intel Core i5 (4 cores), 16GB RAM, and NVIDIA GTX 1650 (4GB VRAM, CUDA 11.8) for GPU-accelerated inference (20-25× speedup over CPU-only) . Docker containerization ensures portability across macOS 11+, Windows 10+, and Ubuntu 20.04+ .

Software Stack:

Backend: Python 3.10, FastAPI 0.104 (async endpoints), SQLAlchemy 2.0 (ORM), PyJWT 2.8 (authentication), ChromaDB 0.4.15 (vector store), LangChain 0.1.0 (RAG orchestration) .

Frontend: Node.js 18, React.js 18.2 (UI components), Axios 1.6 (HTTP client), Material-UI 5.14 (design system), Chart.js 4.4 (metrics visualization) .

Models: Hugging Face Hub models (meta-llama/Llama-3-8b, microsoft/Phi-3-mini-4k-instruct) with 4-bit quantization reducing disk footprint to 4-5GB .

V. RESULTS AND DISCUSSION

A. Experimental Setup

Evaluation employed three benchmark REST APIs: PetStore API (10 endpoints, CRUD operations on pets/stores/users), JSON Placeholder (6 resource types, nested relationships), and a custom Flight Booking API (15 endpoints, multi-step booking workflows) . Baseline comparisons included manual scripting (20 test cases, 8 hours effort), Postman collections (automated but static, 40 cases), and ChatGPT-4 without RAG (cloud-based, 60 cases with 18% hallucination errors).

Metrics: Path coverage (% of API paths executed), parameter coverage (% of parameters tested), defect detection rate (bugs found per 100 test cases), generation time (seconds for 20 payloads), and error rates (% invalid JSON/false positives).

B. Test Coverage Analysis

The proposed RAG-LLM framework achieved 85.2% path coverage and 74.8% parameter coverage on PetStore API (vs. 62% path/55% param for Postman), with 93 cases covering positive/negative/edge scenarios in 138 seconds (Llama 3 8B, GPU-enabled) . Multi-step workflows (e.g., Create User → Login → Book Flight → Cancel) completed successfully in 87% of attempts, compared to 45% for non-RAG ChatGPT-4 (dependency resolution failures).

Coverage Breakdown (PetStore API) :

GET endpoints: 90% (9/10 paths)

POST endpoints: 82% (edge cases for duplicate entries)

PUT/DELETE: 78% (partial coverage for BOLA scenarios)

Authentication flows: 100% (OAuth token, API key methods)

Negative test effectiveness: 68% of generated invalid payloads (malformed JSON, missing required fields) correctly triggered 400/422 responses, exposing 3 undocumented validation bugs in Flight Booking API.

C. Security Vulnerability Detection

Karate DSL integration identified 7 BOLA vulnerabilities in Flight Booking API (unauthorized access to booking IDs via GET /bookings/{id} without token validation) . Comparison with OWASP ZAP automated scans revealed 2 additional API-specific flaws (e.g., mass assignment risk in POST /users) missed by generic scanners . False positive rate: 4.2% (primarily from over-sensitive anomaly detection flagging legitimate conditional responses).

D. Performance and Efficiency

Generation Time: Average 112s for 20 payloads (Llama 3 8B, GPU), 198s (Phi-3 Mini, CPU-only), 45s (ChatGPT-4 API, cloud latency included). Memory footprint: 6.2GB peak usage (model + embeddings + runtime), within 8GB RAM constraint.

Execution Throughput: Sustained 47 parallel API calls/min (httpx async), with 99.1% success rate (network/timeout errors excluded). Dependency resolution overhead: +12% execution time vs. naive parallel (acceptable trade-off for correctness).

Cost Savings: Estimated 70% reduction in QA engineer hours (8 hours manual → 2.4 hours with framework for equivalent coverage), translating to $5,000-7,000 savings per project cycle for mid-sized teams.

E. Usability Evaluation

Qualitative assessment with 5 QA professionals (2-5 years experience) rated the UI 4.2/5.0 for intuitiveness (post-onboarding tutorial) . Pain points: Limited error explanations for LLM generation failures (addressed via enhanced logging in v2), occasional payload preview lag with 50+ cases (React virtualization optimization needed). Accessibility compliance verified via Axe DevTools: 0 critical violations, 2 minor contrast issues resolved.

F. Limitations and Observations

Model Hallucinations: Despite RAG grounding, 3.5% of payloads contained semantically incorrect values (e.g., negative age fields where positive constraints existed) .

Complex Schema Handling: Nested object validation (depth >3 levels) exhibited 12% error rates in schema adherence .

Dynamic Field Prediction: Timestamp/UUID normalization regex required manual tuning per API (planned ML-based pattern learning) .

VI. CONCLUSION

This research successfully demonstrates the feasibility and effectiveness of integrating Retrieval-Augmented Generation with locally deployed Large

Language Models for intelligent, automated API testing . The proposed framework addresses critical gaps in traditional methodologies by achieving 85% path coverage, 95% code-less automation, and 60-80% reduction in manual effort while maintaining sub-150-second latency and 99% uptime on standard hardware . Key innovations include RAG-enhanced payload generation mitigating hallucinations, topological dependency resolution for multi-step workflows, and integrated BOLA security validation via Karate DSL .

Contributions: The work establishes a scalable, privacy-preserving model for AI-driven quality assurance, validated through comprehensive experiments on benchmark APIs . Practical implications include accelerated CI/CD cycles, enhanced defect detection (30-50% improvement), and democratized testing capabilities for non-expert users via intuitive UI design .

Future Directions: Extensions include GraphQL/gRPC protocol support, cloud-native scaling with Kubernetes, federated RAG for multi-tenant collaboration, and self-healing test adaptation via reinforcement learning . Ongoing research explores multimodal LLMs for visual API documentation generation and predictive defect forecasting using historical test analytics .

VII. ACKNOWLEDGMENT

The authors thank Prof. Jondhale D.R. (Project Guide), Prof. Bramhane P.S. (Project Coordinator), and the Department of Artificial Intelligence and Machine Learning, SREIR's Samarth College of Engineering and Management, Belhe, for their invaluable guidance and support throughout this research .

REFERENCES

[1] T. Bardakci, S. Demeyer, and M. Beyazit, "Test Amplification for REST APIs Using Out-of-the-Box Large Language Models," IEEE Software, 2025.

[2] E. M. Pasca, D. Delinschi, R. Erdei, and O. Matei, "LLM-Driven Self-Improving Framework for Security Test Automation Leveraging Karate DSL for Augmented API Resilience," arXiv preprint, 2025.

[3] Isha, A. Sharma, and M. Revathi, "Automated API Testing," International Journal of Computer Applications, vol. 178, no. 36, pp. 1-5, 2018.

[4] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," Advances in Neural Information Processing Systems, vol. 33, 2020.