Automated REST API Black-box Test Generation in Practice: An Experience Report from Industry

Davide Corradini, Seung Yeob Shin, and Domenico Bianculli
University of Luxembourg, Luxembourg
davide.corradini@uni.lu, seungyeob.shin@uni.lu, domenico.bianculli@uni.lu

***Abstract*—Automated black-box test generation for REST APIs has shown promise in research, yet industrial adoption remains limited. This paper presents an experience report on integrating two state-of-the-art REST API test generation tools, RestTestGen and DeepREST, into the quality assurance workflow of a space solutions company, to test a satellite connectivity system's REST APIs comprising 204 API operations across five services implemented in Java, Python, and C++.**

Empirical evaluation on production-like environments revealed high effectiveness, with median 74% operation coverage (151/204 operations) and detection of 21 unique previously unknown faults (e.g., missing input validation, database errors) in four out of five services, confirmed by developers. Both tools executed efficiently on commodity hardware, completing in under 10 minutes for most APIs (median CPU <100%, memory <1 GiB).

The company's software engineers appreciated the tools' ease of use, zero-setup Docker integration, and value for edge/negative scenarios, viewing them as complementary to manual tests rather than replacements. Key lessons include: automated tests complement manual ones by addressing negative scenarios, edge cases, and simple business logic; strict company policies hinder adoption of academic tools; and terminology mismatches between academic tools and practitioner expectations cause confusion. These lessons pave the way to making academic tools more useful in industrial settings and may steer future research on REST API testing in industry.

I. Introduction

In modern software architectures, REST APIs are essential building blocks for system integration, facilitating remote access to functionalities and resources across diverse software systems. Thanks to their system-agnostic interfaces, they interconnect, for example, web applications and mobile applications with their backends, IoT devices with cloud services, and enable microservices architectures. Due to their widespread adoption, REST APIs often provide access to the core functions of *critical* systems [1], [2], implementing sensitive features such as banking transaction processing [3], healthcare management [4], and telecommunications infrastructure management [5].

In this work, we focus on the telecommunications infrastructure management of our industrial partner SES, a space solutions company that operates an integrated multi-orbit satellite network serving a worldwide customer base. At SES, most software systems rely heavily on REST APIs to enable communication between distributed components, manage configurations, and ensure secure and dependable service delivery across their satellite infrastructure.

The use of REST APIs in such sensitive contexts highlights the importance of ensuring their reliability and robustness. Consequently, these APIs typically undergo extensive testing in industry before deployment. Quality assurance (QA) teams develop test cases that cover common usage scenarios and edge cases, aiming to detect defects at the earliest possible stage.

However, writing these test cases manually is labor-intensive and typically limits the range of execution scenarios explored. To address this, the SE research community has recently developed various automated test generation techniques aimed at reducing the testers' workload while also enhancing the quality of REST APIs [6]–[24]. Nevertheless, we have observed only limited evidence of these automated test generation techniques being adopted in industry [25]–[30], with many organizations continuing to rely mainly on manually written test cases [31].

This paper *reports on the experience* of integrating two state-of-the-art test generation tools for REST APIs (RestTestGen [15] and DeepREST [20]), previously evaluated only in research contexts, into the industrial testing process of SES, where test cases were defined manually. Specifically, we targeted a core component-based system (called Open Orbits), exposing a total of 204 API operations over five REST services. After demonstrating the academic tools' functionality to the software QA team, we let the engineers use the tools to test the Open Orbits APIs. We also assessed the effectiveness and efficiency of the tools on industrial-grade APIs, measuring fault detection capability, coverage, and resource consumption.

The empirical evaluation demonstrated the tools' effectiveness, achieving high operation coverage (always ≥70%) and detecting 21 previously unknown bugs (e.g., missing input validation, database errors) across four out of the five REST services, as confirmed by the developers.

Moreover, SES QA engineers reported positively on the tools' ease of use (for example, highlighting the clear documentation and the zero-setup integration), the rapid execution time (in the order of few minutes) to obtain valuable results, the usefulness of the reports generated by the tools, and the value in exploring edge and negative scenarios.

Furthermore, insights from the integration reveal developers' misuse of `5XX` status codes (e.g., `500` instead of `404`), incomplete specifications limiting tool effectiveness, and the need for domain knowledge to generate semantically valid inputs.

From this experience of integrating academic tools in industrial settings, we can draw the following lessons: (i) automatically generated tests complement (and do not replace)

manual tests, with the former addressing negative scenarios, edge cases, and simple business logic, and the latter covering complex business-logic scenarios; (ii) strict company policies may hinder or delay the adoption of academic tools in industrial settings; (iii) there could be a mismatch between the terminology used in the documentation and logs of academic tools and the one adopted by practitioners, which may lead to confusion.

II. Background

This section provides background on REST APIs and automated black-box test generation for REST APIs.

A. REST APIs

REST (REpresentational State Transfer [32]) is the predominant architectural style used for designing web APIs today [33]. REST APIs enable clients to invoke remote functionality and to access and manipulate resources via stateless operations over the HTTP protocol. Resources are uniquely identified by a URI. CRUD (*create*, *read*, *update*, *delete*) operations are mapped to standard HTTP methods such as `POST`, `GET`, `PUT`, or `DELETE`. A request to a REST API consists of an HTTP method applied to a specific URI, often accompanied by input parameters, and results in an HTTP response reporting the status of the processing of the request (e.g., success, client error, or server error) along with optional payload data. The OpenAPI Specification (OAS) standard is widely used to systematically describe REST APIs, formally documenting endpoints, input parameters (with types and constraints), response schemas, and authentication methods.

B. Black-Box Test Generation for REST APIs

Black-box tests for REST APIs consist of sequences of HTTP calls to the system under test (SUT). Effective test generation requires sequencing API calls while respecting *operation dependencies* to avoid failing requests due to client errors and prioritize realistic execution scenarios. In addition, it requires selecting valid and meaningful input values for parameters while respecting syntactic and semantic constraints.

Among the several test generation techniques for REST APIs proposed in the literature, in the following we introduce those implemented by the two academic tools considered in this work: RestTestGen [15] and DeepREST [20], both built on the RestTestGen Framework [34]. We provide a detailed explanation of the industrial partner's reasons for choosing these tools in Section IV-A.

a) RestTestGen: It uses the input and output schema descriptions in the OpenAPI specification of the API under test to build an *Operation Dependency Graph* (ODG for short) which models data dependencies among API operations. An operation is considered dependent on another when it requires as input a parameter that is produced as an output by the other operation. The ODG is used during test generation to determine the order of API operation invocations, prioritizing those with fewer unsatisfied data dependencies. The graph is dynamically updated at runtime based on observed data that satisfy dependencies. Regarding input generation, RestTestGen uses several value sources for parameters: random values generated according to the syntactical constraints specified in the OpenAPI specification, previously observed values, and `example`/`enum`/`default` values declared within the OpenAPI specification. Successful test interactions are eventually mutated to derive negative tests aimed at assessing the resilience of the API in presence of malformed input (e.g., when required parameters are missing or when parameter values do not comply with the constraints in the OpenAPI specification). RestTestGen implements two oracles that analyze the API's responses to reveal defects. The *schema validation oracle* identifies a defect whenever an API response schema deviates from the definition in the OpenAPI specification. The *status code oracle* detects a defect whenever the API responds with a `5XX` status code, indicating an unhandled exception during execution. Additionally, for negative tests, this oracle flags a defect if the API accepts a malformed input (i.e., returns a successful `2XX` status code), indicating inadequate input validation. RestTestGen produces JUnit + REST Assured tests and Postman collections, and reports in HTML and JSON formats.

b) DeepREST: It employs a curiosity-driven deep reinforcement learning agent to explore the API's state space by learning which sequences of operation calls lead to new and testable API states. This technique allows the learning of all kinds of operation dependencies, including logical dependencies not captured in the OpenAPI specification as data dependencies. Simultaneously, it uses experience-driven learning agents to select the most effective input data source, for each parameter of each request, based on past successful interactions. The available input data sources include all those used by RestTestGen, plus an additional one based on a large language model (LLM). This LLM-based source delegates value generation to an LLM, which produces parameter values guided by the parameter name, type, format, and any available natural language description present in the OpenAPI specification. Upon successfully reaching new states, DeepREST intensifies testing by mutating past successful API requests to increase coverage and fault detection, deriving further positive but also new negative tests. DeepREST implements the same status code and schema validation oracles as RestTestGen.

III. Industrial Deployment Context

This section describes the industrial deployment context at SES, including the systems under test (Open Orbits) and the existing testing practices.

A. Context

As part of a long-term partnership with SES focused on research, knowledge and technology transfer, discussions with the head of the software QA team revealed a need for advanced automation in the testing process, including assessment of next-gen, AI-based test generation tools. With REST APIs being widely used across all SES systems, this software type appeared as a promising testbed for introducing such automation tools.

Therefore, we decided to trial the integration of automated test generation tools for REST APIs into SES's software

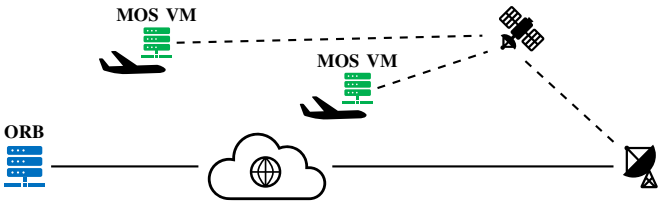

Figure 1. End-to-end deployment architecture of Open Orbits.

development process. The initial trial was set to apply the test generation tools only to one system under test (SUT), with potential expansion to other SUT in case of success.

The software QA head selected Open Orbits, which we detail below, as the target SUT because it would particularly benefit from automated REST API test generation, given its extensive API surface and manual testing processes. Additionally, the two teams involved in its development were relatively new and open to adopting innovative tooling.

B. The Systems Under Test

Open Orbits is SES's multi-orbit, open-architecture in-flight connectivity network. It seamlessly integrates the company's geostationary and medium-earth-orbit satellites with regional partners, providing high-speed internet to airline passengers.

Figure 1 depicts the end-to-end deployment architecture of Open Orbits. It consists of the ground-based subsystem called ORB (cloud-deployed) and multiple instances of the aircraft-based subsystem called MOS. Network operators use ORB to configure network settings (interface parameters, country-specific internet availability, satellite beam priorities), schedule software updates, collect statistics, and monitor aircraft systems health. Each aircraft adhering to Open Orbits hosts its own MOS virtual machine instance—connected to dedicated hardware such as antennas and modems—which pulls configurations from the single ORB instance via satellite links and delivers internet connectivity to passengers according to the defined configurations. These critical tasks are cooperatively handled by five REST services: one in the ground-based ORB subsystem, and four in the aircraft-based MOS virtual machines. Although Open Orbits includes other software components, this section focuses on its REST APIs, which are this paper's primary subject.

Two distinct teams of software engineers develop the ORB and MOS subsystems. Each team follows its own internal development (and testing) processes, tailored to the specific environment they work in.

1) ORB: The ORB subsystem exposes a single REST API—which serves as the core of the Open Orbits system—for network configuration. In production, this API is deployed in a containerized cloud environment. It is implemented in Java using the Spring Boot framework, and relies on a Postgres database for persistence; it exposes 93 operations, all documented in an OpenAPI specification. The OpenAPI specification is automatically generated from source code using the *springdoc-openapi* library, with developers adding annotations to enrich the documentation.

Table I
THE REST API OF OPEN ORBITS.

API Name	# Ops.	PL	REST Framework
ORB	93	Java	Spring Boot
MOS Core	66	C++	Drogon
MOS Update Manager	19	Python	FastAPI
MOS Health Checker	19	Python	FastAPI
MOS Stats Manager	7	Python	FastAPI

2) MOS: The MOS subsystem integrates both native services and containerized microservices, exposing a total of four REST APIs: a core C++ service for hardware control (e.g., prioritizing satellite beam selection) implemented with the Drogon framework, and three Python-based services (implemented using FastAPI) for managing software updates, statistics collection, and continuous health monitoring. OpenAPI specifications for these APIs are manually written and maintained.

Table I summarizes the characteristics of Open Orbits's five REST APIs. For each API, we report the number of exposed API operations as a metric of their size [2], [35], as well as the programming language (PL) and the REST framework used for its implementation.

C. Existing Testing Process

The two Open Orbits development teams follow an Agile-DevOps workflow with continuous integration (CI) and automated testing at its core. REST API tests are manually written by QA engineers to cover all business features of the system, and executed via automation frameworks. The teams run tests iteratively with local runners during sprints, then integrate them into CI pipelines for automated execution after every commit or pull request. SES enforces a strict 90% test coverage target: pipelines fail and block merges if such a target is not reached.

Given their distinct runtime environments and heterogeneous technology stacks, ORB and MOS teams maintain tailored testing processes, as described below.

1) ORB: The ORB test suite currently consists of more than 500 API tests, developed in about one year. REST API tests are written using the REST Assured library and encapsulated within JUnit test classes for automated execution. Direct REST API invocations also occur in end-to-end tests, which are authored using the Playwright framework, using JavaScript. Postman is used for occasional exploratory testing.

Dedicated test cloud environments, isolated from production, support ORB testing. Moreover, engineers can run ORB as a standalone instance on their local machines for testing purposes.

2) MOS: The MOS test suite currently consists of around 200 tests targeting the four REST APIs, developed in about one year. The MOS team writes all REST API and end-to-end tests—including those targeting the C++ API—in Python using the Robot Framework, which features a human-friendly syntax. Postman supports occasional exploratory testing.

Due to the MOS virtual machine's complex deployment, integrating with aircraft hardware like satellite modems and antennas, SES maintains a ground-based test lab replicating the

exact hardware and software stack, connected exclusively via real satellite links to mirror production conditions. Engineers access these shared lab machines for test campaigns.

IV. Tools Selection and Integration

This section details the selection and integration of the REST API test generation tools into SES's testing workflows for the Open Orbits APIs.

A. Tools Selection

SES defined selection criteria for REST API test generation tools that would integrate seamlessly with their existing testing workflows. The following factors guided this selection:

- **Technology-agnostic, black-box tools.** To ensure applicability across SES's heterogeneous REST APIs (written in Java, Python, and C++), the company demanded for tools that generate black-box test cases and operate solely through the HTTP interface of APIs, without requiring access to source code or depending on the underlying technology stack.
- **Minimal setup effort.** To minimize engineers' effort and integration overhead, SES required tools that execute without extensive configuration, additional dependencies, or complex environment setup.
- **Compatibility with current testing practices.** For seamless integration, SES required tools to leverage existing artifacts (e.g., OpenAPI specifications) without additional inputs, and produce outputs compatible with current workflows (e.g., JUnit + REST Assured test suites and Postman collections).
- **Ease of extensibility.** For future evolution, SES valued tools with extensible architectures that enable adding new output formats or custom test generation strategies with minimal engineering effort.
- **Documentation consistency checking.** Beyond fault detection, SES required tools capable of detecting discrepancies between OpenAPI specifications and actual API implementations to maintain documentation accuracy.
- **Reliability and robustness.** Since the selected tools would be integrated into daily QA routines, SES prioritized mature, actively maintained, and stable implementations that avoid crashes or inconsistent behavior.

Based on these criteria, after a review of the state of the art conducted together with the first author, SES selected RestTestGen and DeepREST. They ensure *technology-agnostic* applicability through black-box HTTP-only operation, independent of the technology stack used to implement the SUT; require *minimal setup effort* through a simple, minimal configuration and by providing Docker images for the building and execution stages with no other dependencies; provide *compatibility with current testing practices* at SES using OpenAPI inputs and generating JUnit + REST Assured tests and Postman collections; offer *ease of extensibility* via the extensible RestTestGen Framework [34]; enable *documentation consistency checking* through schema validation oracles; and demonstrate *reliability and robustness* through active maintenance since 2020. The shared RestTestGen framework across the two tools also ensures consistent CLI, inputs, outputs, and workflows, facilitating adoption by the QA engineers and tool interchangeability.

B. Tools Integration

The first step towards the tools' integration involved going through a security assessment of the tools performed by SES's security team, to ensure compliance with the company's strict data protection policies. The security team checked the documentation from the tools' official GitHub repositories and performed a review of the source code, focusing on potential side effects and data handling practices.

After obtaining security clearance, the second step involved setting up a dedicated isolated environment hosting a test instance of the ORB API, separate from production. This was done to prevent the tool's potential irreversible side effects (e.g., data deletion) from affecting the live system.

The integration process began with a training session (1-hour demonstration) for the ORB QA team, carried over by the first author. This session—focused on RestTestGen and based on a dummy REST API—covered tool configuration, Java and Docker execution, interpretation of CLI log messages, as well as analysis of generated reports (HTML, JSON) and test cases (JUnit, Postman).

After the training, a designated team delegate was then tasked to execute RestTestGen on the ORB API using its default configuration. During this phase, the first author remained available for support. However, the delegate successfully ran RestTestGen independently, relying solely on the official README and the information from the training session—without requiring any support. The tool successfully revealed a few faults in the ORB API not caught by existing tests.

Based on this initial, positive outcome, SES decided to extend tool usage to the MOS team as well. The first author gave the same 1-hour demo session to the MOS QA team. A team delegate was tasked to run RestTestGen on all four MOS APIs. Unlike ORB, MOS APIs depend on real hardware (e.g., modems) connected to satellites, making the creation of a dedicated isolated environment infeasible. Instead, the delegate used the existing MOS test lab—which has a recovery system that restores the system to its pre-test state after execution—allowing safe testing across all four MOS APIs despite potential data deletion by the tests generated by RestTestGen.

The next step involved integrating DeepREST, which uses deep reinforcement learning for API exploration and an LLM for generating realistic input parameter values. Although DeepREST employs a fundamentally different testing strategy than RestTestGen, both tools share identical usage patterns (configuration, CLI, logs), enabling the practitioner delegate to execute DeepREST independently without support.

The only usage difference from RestTestGen lies in LLM integration. By default, DeepREST invokes the LLM on-the-fly during testing, caching generated values in a persistent dictionary for reuse. To eliminate the need for engineers to configure the LLM or require LLM access during testing (which is forbidden by SES's security policies), the first author pre-generated the complete dictionary for all Open Orbits APIs

using a utility bundled with DeepREST; this dictionary was then used during testing without any live LLM calls. Specifically, the dictionary was generated using a local Qwen 3 model (14B parameters) running in LM Studio on an offline machine. This model was chosen as it was the top performer in Yang et al.'s report [36] at the time and met our machine's resource limits.

V. Industrial Application - Empirical Study

To complement the report on the industrial adoption of RestTestGen and DeepREST, in this section we present an empirical study on the application of these tools to the five Open Orbits REST APIs. The primary objective of this study is to empirically evaluate the effectiveness and efficiency of the two black-box REST API test generation tools when applied to industrial-grade APIs. These APIs differ from academic or open-source counterparts, which typically feature toy examples and simple implementations, as they are production-ready services and part of enterprise systems characterized by high scalability, robust security measures, real-world dependencies (e.g., live databases), strict regulatory compliance, and heavy traffic-handling capabilities. Moreover, such services are typically subject to extensive testing prior to deployment. This study contributes to the field as one of the first examples of applying these black-box testing tools to real-world industrial-grade APIs.

To guide our empirical evaluation, we formulated the following research questions (RQs).

RQ1: *What is the effectiveness of RestTestGen and DeepREST when applied to industrial-grade APIs?*

With this RQ, we evaluate how effective the tools are on industrial APIs, measuring operation coverage and detection capability of previously unknown faults via generated tests.

RQ2: *What is the efficiency of RestTestGen and DeepREST, in terms of resource consumption when applied to industrial-grade APIs?*

With this RQ, we assess the resource usage (execution time, CPU usage, and memory consumption) of the tools when applied to industrial-grade APIs, to understand their applicability in real-world industrial QA workflows.

A. Experimental Settings

We conducted the empirical evaluation on the Open Orbits REST APIs described in Section III-B and listed in Table I, using RestTestGen and DeepREST in their default configurations, i.e., the same settings adopted and recommended by the tools' authors in their original studies. The APIs were initialized with real-world data snapshotted from production.

Both tools ran in Docker containers, as per official documentation. The experiments were conducted on a QA team delegate's Apple MacBook Air M3 with 16 GB RAM running macOS Tahoe 26.2 and Docker 29.1.5.

B. RQ1: Effectiveness

1) Methodology: The tools were configured with a test budget defined as the maximum number of HTTP requests, set to $200 \times O$, where O denotes the number of exposed operations per API—a threshold shown sufficient for convergence in prior REST API testing studies [20]. This budget ensures a fair comparison, as the two tools generate the same number of requests per API. Results are not compared across APIs, which have different budgets due to varying numbers of operations. In our context, we believe a time budget is unsuitable, as execution duration is primarily dominated by satellite network latency and API response time, rather than tool computation. Hence, imposing a time budget would confound environmental delays with the tools' actual processing costs.

To account for non-determinism (e.g., random input generation and reinforcement learning in DeepREST), each tool-API pair in our experiments was run independently 10 times, resetting the APIs to their initial state (real-world data snapshot) after each execution.

We measured the following metrics:

- **Request outcomes.** Number and ratio of successful requests (`2XX` status codes), client errors (`4XX` status codes), and server errors (`5XX` status codes), relative to the total request budget. These metrics are computed from the HTTP interaction history logged by tools. While not a direct effectiveness measure (e.g., valid requests to read-only endpoints may not exercise extensive API logic or trigger faults), it correlates with effectiveness for tools like RestTestGen and DeepREST that employ fair test generation strategies without artificially boosting `2XX` outcomes. Overall, these metrics provide insight into test generation capabilities and the difficulty that tools encounter in generating successful or fault-triggering requests.
- **Successful operation coverage** (or **operation coverage** for short). An API operation is considered successfully covered if at least one request targeting that operation is accepted by the API with a successful `2XX` status code. We measure operation coverage both as an absolute count of unique API operations exercised successfully and as a percentage value defined as the ratio between covered operations and the total number of operations exposed by the API. This metric is computed and reported by the tools in their coverage reports.
- **Unique faults detected.** Distinct server errors (`5XX` status codes), where sufficiently similar errors are bucketed and counted as a single unique fault. This metric is computed from the HTTP interaction history logged by the tools, using a response-payload similarity algorithm proposed in previous work [19]. We manually adapted the original algorithm (e.g., by customizing the Jaccard similarity threshold and the error text preprocessing for the Open Orbits APIs) to ensure the correctness of bucketing for the five Open Orbits APIs.

2) Results: Table II reports the results for request outcomes. Given that, for each tool-API pair, the request outcomes showed minimal variance across 10 repetitions, the table reports averaged values from those runs. In general, we observe the tools generate requests that, in the majority of cases, are either accepted by the API (`2XX`) or rejected as client errors (`4XX`). RestTestGen generates 39% of `2XX` interactions (49% of `4XX` interactions), whereas DeepREST generates 55% of `2XX` interactions (38% of `4XX` interactions). We believe that

Table II
RQ1 - Results for Effectiveness: Request Outcomes ("Ops." denotes the number of API operations).

API	Ops.	Budget	RestTestGen						DeepREST					
			2XX		4XX		5XX		2XX		4XX		5XX	
			abs	%	abs	%	abs	%	abs	%	abs	%	abs	%
ORB	93	18600	3481	19	11279	61	3840	21	7807	42	8666	47	2128	11
MOS Core	66	13200	6120	46	5740	43	1340	10	7811	59	4514	34	875	7
MOS Update Manager	19	3800	1824	48	1976	52	0	0	2499	66	1301	34	0	0
MOS Health Checker	19	3800	3523	93	268	7	9	0	3394	89	392	10	14	0
MOS Stats Manager	7	1400	820	59	578	41	1	0	934	67	465	33	1	0
Total	204	40800	15768	39	19841	49	5190	13	22445	55	15338	38	3018	7

Table III
RQ1 - Results for Effectiveness: Operation Coverage and Fault Detection. ("Op. Cov. (abs)" and "Op. Cov (%)" denote absolute and percentage Operation Coverage, "Unq. Faults" denotes Unique Faults).

API	Ops.	RestTestGen						DeepREST					
		Op. Cov. (abs)		Op. Cov. (%)		Unq. Faults		Op. Cov. (abs)		Op. Cov. (%)		Unq. Faults	
		med	max	med	max	med	max	med	max	med	max	med	max
ORB	93	67	67	72	72	9	10	67	71	72	76	10	11
MOS Core	66	46	46	70	70	6	6	46	46	70	70	6	6
MOS Update Manager	19	15	15	79	79	0	0	14	15	74	79	0	0
MOS Health Checker	19	18	19	95	100	1	3	18	19	95	100	1	3
MOS Stats Manager	7	5	6	71	86	1	1	6	6	86	86	1	1
Total	204	151	153	74	75	17	20	151	157	74	77	18	21

the higher proportion of `2XX` and the lower proportion of `4XX` and `5XX` observed with DeepREST are due to its deep-learning-based exploration algorithms, which favor the exploration of new execution scenarios and penalize repetitive invocations of API operations known to lead to server errors, thus suggesting a more effective exploration strategy. Conversely, RestTestGen tends to repeatedly exercise operations that are harder to satisfy, wasting test budget on `4XX` interactions.

Requests leading to `5XX` status codes correspond to only 13% of those generated by RestTestGen and 7% of those generated by DeepREST. For some APIs, the tools have difficulty generating fault-triggering requests, as in the case of MOS Update Manager, for which we observe no `5XX` responses, or MOS Health Checker and MOS Stats Manager, for which only a minimal number of requests result in `5XX` responses. For ORB and MOS Core, instead, we observe a substantially higher proportion of `5XX` responses, up to 13% when tested with RestTestGen. The fact that only a few `5XX` responses are observed for MOS Health Checker and MOS Stats Manager suggests that the underlying faults are harder to trigger, possibly because they require very specific inputs (hard to guess by tools) that drive execution into deep paths in the CFG. By contrast, the high rate of `2XX` interactions with MOS Health Checker suggests that the API requires few parameters or parameters with a broad valid input range, which in turn makes it easier to test.

Table III summarizes operation coverage (absolute and percentage) and fault detection. For operation coverage and fault detection, we report median and maximum values computed over the 10 repetitions, as these metrics better capture the distribution within the relatively small numeric range (0–100) and highlight subtle variations in performance. Minimum values are omitted to avoid redundancy, as they aligned with the median in most instances, or were just lower by one unit.

Regarding operation coverage, the two tools exhibit comparable effectiveness, with a slight advantage for DeepREST. Both achieve the same median number of covered operations (151 out of 204, or 74%), with RestTestGen covering one additional operation in MOS Update Manager and one fewer in MOS Stats Manager. However, the maximum number of covered operations is 153 (75%) for RestTestGen and 157 (77%) for DeepREST. This result suggests that DeepREST can cover more operations, but only in a subset of its executions, indicating that its DRL-based exploration has the potential to learn more effective testing strategies. We note that this potential is not always realized due to the stochastic nature of learning and the limited budget in our setting, as shown by the gap between the maximum and median values in the table. The operation coverage achieved by RestTestGen and DeepREST on the Open Orbits APIs exceeds the effectiveness reported in the original DeepREST paper [20] (which also reports results for RestTestGen as part of a comparative study), where the evaluation was conducted on open-source APIs. Although we do not provide quantitative evidence that the Open Orbits APIs are more complex, our manual analysis suggests that this is the case (with more operations, stronger dependencies among operations, more parameters, and tighter constraints). We attribute the improved effectiveness to the rigorous OpenAPI specifications produced at SES which, although not complete (e.g., many parameters

lack examples), offer precise schema definitions that facilitate the generation of syntactically valid requests.

Regarding fault detection, both tools were able to identify several previously unknown faults in four out of the five Open Orbits APIs, with MOS Update Manager being the only API for which no faults were identified. In total, RestTestGen and DeepREST revealed 17 and 18 unique faults, respectively, with peaks of 20 and 21 unique faults across the 10 executions. Notably, DeepREST detected all faults found by RestTestGen, plus one additional fault in the ORB API.

The case of MOS Health Checker is peculiar: in most runs, both tools identified only a single fault, whereas in a few runs they were able to expose up to three unique faults, suggesting the presence of hard-to-trigger faults that require specific input conditions or longer interaction sequences to be activated.

> **Answer to RQ1:** *RestTestGen and DeepREST proved effective on the five industrial-grade Open Orbits APIs, achieving a median operation coverage of 74% (covering 151 out of 204 operations) and revealing previously unknown faults in four out of five APIs, with up to 21 unique faults in the best executions.*

C. RQ2: Efficiency

1) Methodology: We applied the same experimental methodology as for RQ1, but measured efficiency-related metrics. Specifically, we collected:

- **Execution time.** Total execution time as reported in the tool output. Although we did not impose a time budget, measuring execution time can still provide insights into the practical cost of running the tools at SES.
- **CPU usage.** Median and peak CPU utilization from samples collected every 2 seconds via `docker stats`, expressed as a percentage where 100% corresponds to one fully utilized core and values above 100% indicate usage of multiple cores.
- **Memory consumption.** Median and peak memory usage from samples collected every 2 seconds via `docker stats`, expressed in MiB.

2) Results: Table IV reports the efficiency results. For each tool applied to each API across the 10 repetitions, the table shows the minimum, median, and maximum execution time (in seconds), the median and maximum CPU usage (expressed as a percentage) over all samples collected every 2 seconds in the 10 repetitions, and the median and maximum memory usage (expressed in MiB) over the same samples.

Execution times are generally low: in most cases, they are below ten minutes, with a minimum of 92 s for RestTestGen on MOS Stats Manager. The only notable exception is MOS Health Checker, for which we observe longer execution times, often exceeding half an hour. Overall, RestTestGen and DeepREST exhibit similar execution times.

We initially expected execution time to correlate with the request budget (i.e., the more requests the longer to generate and execute them), but this was not always the case in our experiments, mainly for two reasons. First, network latency plays a key role: APIs hosted on local networks complete quickly despite large budgets and high request volumes (e.g., median of 352 s for RestTestGen and 261 s for DeepREST on the ORB API with <1 ms RTT), whereas remote APIs over remote links (e.g., satellite) take substantially longer. Second, APIs with time-consuming processing—such as deep health checks that delay responses for seconds, like in MOS Health Checker—extend execution time, as tools primarily wait for replies. As anticipated, the duration of a testing session is therefore largely dominated by network and API processing times rather than by the tools' internal processing. Nonetheless, we note that reporting execution time remains important, as it provides an indication of the order of magnitude of the effort required to use these tools in an industrial context.

Regarding CPU usage, the median value remains below 100% for all executions (except for DeepREST on the ORB API), indicating that, on average, the tools use less than one CPU core. However, we observe relevant CPU usage peaks, up to 475.9% for RestTestGen on MOS Core and 743.3% for DeepREST on the ORB API. The higher CPU consumption can be attributed to the very short response times of the ORB API over the local network, which enable a higher request throughput and cause the tools to process requests almost continuously, with virtually no idle periods. For the MOS Health Checker API, the median CPU usage is particularly low, approaching 0.2% for both tools; we attribute this to the fact that, for most of the time, the tools are idle waiting for API responses and thus consuming no CPU resources.

Memory usage is also moderate: in most cases, the median memory consumption is below 1 GiB for both tools, with an absolute peak of 4569 MiB recorded for DeepREST when testing MOS Core, and a peak of 3049 MiB for RestTestGen, when testing ORB. Overall, RestTestGen appears to be less memory demanding than DeepREST.

> **Answer to RQ2:** *RestTestGen and DeepREST demonstrated good efficiency on the industrial-grade Open Orbits APIs, with relatively short execution times, median CPU usage below 100% for almost all executions, and median memory usage typically below 1 GiB for both tools. Peak memory consumption reached 3049 MiB for RestTestGen when testing ORB, and 4569 MiB for DeepREST when testing MOS Core. These resource profiles indicate that both tools can be executed on commodity hardware, such as company-provided laptops, and are therefore suitable for integration into real-world industrial QA workflows.*

D. Threats to Validity

Internal Validity. The effectiveness results may be influenced by the specific configuration of the tools. We mitigated this by using the default configurations recommended by the tool authors, which are intended as optimized settings for typical use cases and reduce the risk of misconfiguration. The bucketing algorithm for identifying unique faults was manually adapted, which may introduce some subjectivity; however, the first author applied it consistently across all APIs, and developers confirmed the uniqueness of the resulting faults.

Table IV
RQ2 - Results for Efficiency: Execution Time, CPU Usage, and Memory Consumption.

API	RestTestGen							DeepREST						
	Time (s)			CPU (%)		Mem. (MiB)		Time (s)			CPU (%)		Mem. (MiB)	
	min	med	max	med	max	med	max	min	med	max	med	max	med	max
ORB	275	352	374	94.7	299.7	1763	3049	245	261	288	202.6	743.3	1288	3025
MOS Core	405	453	540	18.4	475.9	462	1427	432	764	1900	56.9	546.2	1677	4569
MOS Update Manager	227	237	245	5.6	133.1	272	344	284	297	309	32.1	402.0	424	523
MOS Health Checker	1864	1938	2028	0.2	191.6	418	845	1433	2161	2712	0.2	143.2	644	968
MOS Stats Manager	92	122	168	13.3	186.4	401	1905	87	138	193	26.0	295.8	696	3261

External Validity. Our study focuses on five APIs from a single organization within the satellite telecommunications domain. While these APIs are production-grade and implemented in heterogeneous technology stacks (Java, Python, C++), findings may not generalize to APIs in other domains or those adopting different architectural patterns. The Open Orbits APIs were initialized with production data snapshots, but testing occurred in isolated environments rather than live production systems, potentially affecting the realism of detected faults.

Construct Validity. Operation coverage and fault detection are established metrics for black-box REST API testing, but they have known limitations. Operation coverage ignores differences in operation complexity or importance, and code coverage was infeasible since the tested APIs could not be instrumented. We also did not study faults detected by existing manual test suites but missed by the tools (i.e., false negatives), since we only had access to the current API versions, which already included fixes for older bugs. Furthermore, the request outcome metric is only an indirect indicator of effectiveness, as it depends on both tool behavior and API characteristics. To mitigate these issues, developers validated all detected faults, we combined metrics with practitioner feedback, and we discussed the indirect nature of request outcomes in our analysis.

Conclusion Validity. Non-determinism in the tools (e.g., random input generation and reinforcement learning) may affect the reliability of coverage and fault detection metrics. We mitigated this by running 10 independent repetitions per tool-API pair and reporting median and maximum values to capture both typical and best-case performance. Because this is an industrial case study involving five REST services, we did not apply formal statistical tests, so observed differences between tools (e.g., DeepREST's higher peak coverage) should be interpreted as indicative trends rather than statistically proven superiority.

VI. Discussion

In this section, we review the feedback received from the engineers, discuss insights and lessons learned from the tools integration, and outline future research avenues as determined by this industrial experience.

A. Engineers' Feedback

We collected feedback from six engineers across the ORB and MOS QA teams during initial demonstrations, early hands-on usage, and throughout the collaboration period while the tools were actively used. The comments and questions gathered during these interactions are summarized below.

a) Feedback on tool adoption: Following successful adoption, both the MOS and ORB teams have decided to incorporate RestTestGen and DeepREST into their workflows. RestTestGen is used for rapid fault detection when adding new features or significantly changing existing API operations. It generates initial tests in JUnit/Postman formats and completes execution in a few minutes, often revealing bugs before manual test development begins. DeepREST is applied less frequently for deeper exploration of complex scenarios, given its longer execution times. The engineers leverage these tools for exploratory testing, especially after changes or new implementations, complementing their manual testing processes. Future plans include integrating RestTestGen and DeepREST into the Open Orbits CI/CD pipeline to automatically detect `5XX` errors and schema violations. This requires adding JUnit XML report support to the tools to enable pipeline compatibility.

b) Ease of use of the tools: The engineers found the tools straightforward to execute, appreciating both the clarity of the README documentation and the 1-hour initial demonstration, which effectively conveyed essential setup information. This ease also resulted from minimal configuration requirements and Docker-based execution requiring no additional dependencies.

c) Usefulness of the generated reports: The engineers found the generated reports highly useful, appreciating the intuitive GUI and filtering features that facilitate locating interesting HTTP interactions. However, they noted that the large volume of requests in the reports can clutter the user experience.

d) Limited comprehensibility of the generated test: The tools generated thousands of test cases, but engineers noted that only a small subset were particularly insightful, making it challenging to identify the most valuable ones. The generated JUnit tests use non-descriptive method and variable names like `test0`, `var1`, and lack comments, hindering code comprehension. Moreover, among the many generated tests, some include requests that engineers find useless (e.g., they execute "irrelevant scenarios") and repetitive, unlikely to what they would write manually. These requests are required for the tools' systematic API exploration, yet appear nonsense to end users.

e) Generated tests as a solid starting point: The engineers remarked that running the tools during new API operation implementation or evolution quickly uncovers faults at early stages of the development. Some generated JUnit tests that

resulted in failures have been incorporated into regression test suites, while also have inspired manual tests for similar scenarios. This enables engineers to focus on complex cases, as the tools automatically cover many simpler scenarios that would otherwise require substantial manual effort.

f) Value for edge cases and negative scenarios: The engineers valued the tools' ability to explore edge cases and negative scenarios, which complement the extensive manual tests already covering positive/happy paths.

g) Characterization of fault-triggering input: The engineers appreciated the tools' automation of test interaction generation and oracle-based evaluation to reveal defects. They noted that the only missing piece for full automation value would be fault characterization—specifically, identifying *which* input elements triggered faults and *why* (e.g., constraint violations or unexpected data flows)—to streamline debugging.

B. Insights from Tool Integration

Beyond the engineers' feedback, our experience has provided technical insights into REST API quality and tool applicability in industrial settings as follows.

a) Detected faults and examples: The tools identified diverse faults across the Open Orbits industrial-grade APIs, including out-of-bounds array access errors, database-related errors, and invalid enum parameter values outside defined ranges. Additional faults include missing type validation on request payloads and generic `500` server errors lacking diagnostic messages, which SES developers are currently investigating.

b) Misuse of `5XX` status codes: Developers sometimes misused `500` status codes instead of appropriate alternatives. For instance, we observed "resource not found" errors returning `500` instead of the standard `404`, and handled exceptions like type validation failures raising `500` rather than `400`. While these trigger false bug detections, the reports remain valuable by highlighting common anti-patterns that compromise API quality and HTTP standard compliance.

c) Specification quality affects tools effectiveness: Completeness and correctness of the OpenAPI specifications strongly affect the effectiveness of black-box test generation tools. Incomplete specifications often omit parameter constraints and example values, limiting the effectiveness of the tools using them as a starting point for test generation. After enhancing OpenAPI specifications with realistic example values, engineers observed improved performance of the tools in terms of covered operations. This performance improvement was observed after executing the experimental campaign presented in Section V.

d) Domain knowledge essential for valid input: Valid and business-logic-related inputs are required to explore relevant execution scenarios within the API under test or simply to generate individual valid requests. Generating such inputs requires domain knowledge that extends beyond syntactic constraints in the OpenAPI specifications. Although DeepREST's LLM-based value generator leverages parameter names, HTTP paths, and natural language descriptions from the OpenAPI specification, it struggled with Open Orbits specifications because not all parameters and operations have descriptions, and existing ones are short or lack useful contextual information. This limitation led to some operations remaining uncovered.

e) Technical limitations of tools: While the tools achieved good operation coverage (always ≥70%), many of the operations remained uncovered due to a technical limitation in RestTestGen framework, which does not support the generation of `multipart/form-data` requests. The framework's developers acknowledged this limitation, admitting that the feature was omitted during the development of the tools since this content type was deemed infrequent. However, Open Orbits APIs use this content type frequently, thus reducing tools' operation coverage.

f) Indirect fault identification: An engineer discovered a hidden API misconfiguration by reviewing the HTML report generated by RestTestGen. An operation with no input parameters unexpectedly returned a `4XX` error, despite not accepting inputs that could cause client-side failures, revealing a misconfiguration that had gone unnoticed for a while. This demonstrates how report inspection can uncover issues beyond automated fault detection, complementing the tools' oracles.

C. Lessons Learned

Based on the engineers' feedback and our technical insights, we have identified three key lessons for adopting academic REST API testing tools in industry. Although they address practical barriers and integration strategies observed at SES, we believe they are applicable in other companies with a similar testing process, procedure, and corporate policies.

a) Automated tools complement manual testing: RestTestGen and DeepREST proved effective on Open Orbits APIs, detecting previously unknown faults across four out of five services. Nevertheless, generated tests cannot fully replace manually written test suites, which remain essential for comprehensive test suites. At SES, manual tests are mandated to achieve at least 90% of code coverage, a threshold difficult for automated tools to reach consistently. The two approaches complement each other effectively, with automation accelerating edge-case and negative scenarios exploration while manual tests ensuring critical path coverage.

b) Overhead from zero outage policies: Due to the critical infrastructure it manages, SES implements a strict *Zero Outage Initiative* with best practices, protocols, and procedures (including multi-person reviews of changes and comprehensive action tracking) to ensure service continuity and minimize disruptions. While these guarantee reliable operations and rapid recovery, they create significant overhead for uncommon procedures like integrating academic testing tools into development workflows.

c) Academic terminology might confuse engineers: Given that the tools, as research prototypes, were initially designed for academic use, their README, log messages, and reports use a terminology rooted in the research literature, which may sound unfamiliar in industrial contexts. For instance, "oracle", a standard academic term for automated fault detection, confused engineers, who mapped it to concepts as "assertions" or "response validation checks". Similarly, DeepREST logs include references to reinforcement learning concepts like

"reward" and "action", which confused engineers. In a few cases, engineers expressed confusion over this terminology during tool demonstrations and support calls. Adapting documentation, logs, and reports to specific industry contexts would significantly enhance usability and adoption of academic tools.

D. Future Research Avenues

Based on the engineers' feedback and our first-hand experience of integrating academic tools in an industrial context, we have identified the following research avenues.

a) Improving test generation algorithms: Current test generation approaches struggle with APIs requiring domain-specific inputs or complex operation dependencies not captured in OpenAPI specifications. Future work could integrate domain knowledge via configurable dictionaries or fine-tuned LLMs trained on domain-specific examples, boosting the generation of semantically valid requests. Additionally, identifying business logic dependencies and request flows would enable generation of realistic sequences for complex execution scenarios.

b) Improving comprehensibility of generated tests: State-of-the-art REST API testing tools generate tests with generic names (e.g., `test0`, `var1`) and opaque sequences, hindering their comprehensibility for users. Future work could explore LLM-based naming/renaming (e.g., inferring intent from the source code of generated tests).

c) Characterization of fault-triggering input: To achieve full automation in REST API testing—beyond generating test scenarios and oracle-based behavior assessment—the crucial next step is pinpointing exactly which input components and their specific characteristics triggered the detected fault. Future work could employ differential analysis on passing vs. failing inputs (e.g., highlighting mutated fields causing `5XX` errors), invariant checks, or static/dynamic slicing to trace root causes and facilitate developer fixes.

d) Test suite minimization: State-of-the-art REST API testing tools generate thousands of black-box tests rapidly, making it hard for engineers to find the most useful ones among the many. Future work could focus on test suite minimization or ranking techniques to deliver concise, high-impact test sets for efficient review and maintenance. This would involve customizing minimization strategies for black-box REST API test suites, such as deciding which tests to select without source code access for structural coverage metrics or visibility into internal execution traces to preserve complex request flows.

VII. Related Work

Research on test generation for REST APIs has been active for several years [37]. However, there are still very few documented cases of these techniques being adopted in industry.

Karlsson et al. [10], in their paper introducing QuickREST, validate their tool on both open-source and industrial APIs, the latter by their partner ABB. However, their work primarily focuses on the technical approach implemented in QuickREST and its evaluation, rather than presenting an experience report on its industrial adoption. Martin-Lopez et al. [38] evaluated their tool RESTest [11] on a benchmark of 13 industrial-grade APIs without direct industrial collaboration, using mainstream web services such as Spotify and Stripe as test subjects. While they report insightful results on the types of faults found in these industrial APIs, unlike our study they did not work with an industrial partner and therefore do not provide insights from engineers. Liu et al., the authors of Morest [18], presented an experience report [25] in collaboration with Huawei. Their tool uncovered 83 new bugs across ten web APIs, all of which were confirmed by developers. Similarly to our study, they observed that automatically generated tests reveal previously unknown defects, especially by exploring corner cases. EvoMaster white-box [7] has been adopted at Meituan, where its RPC extension (distinct from REST) identified 41 confirmed faults that engineers subsequently fixed [26], [27]. Subsequent user studies further validated its practicality: in the initial study, practitioners retained 100% of the generated tests in their CI pipeline; in the follow-up, retention reached 84% [28]. EvoMaster black-box has been adopted at Volkswagen [29], [30]. Engineers considered its outputs as effective starting points for test suites, accelerating initial creation but needing expert review, pruning, and refinement for complex APIs. It detected production defects missed by manual regression tests.

Overall, prior industrial studies have focused on other REST testing tools, and only a few have explored industrial adoption of automated REST API test generation. Ours is the first report on the industrial use of RestTestGen and DeepREST on production-grade APIs, combining quantitative results on a critical satellite system with qualitative feedback and lessons from engineers.

VIII. Conclusion

In this paper, we have reported on the successful industrial adoption of two black-box REST API test generation tools, RestTestGen and DeepREST, for testing a satellite connectivity system, which exposes a total of 204 API operations across five services implemented in Java, Python, and C++.

Our empirical evaluation showed high effectiveness, with a median operation coverage of 74% (151/204 API operations) and detection of 21 unique previously unknown faults in four services, alongside efficient resource usage suitable for execution on commodity hardware such as typical QA engineer laptops. The engineers valued the tools' ease of use, quick execution (mostly under 10 minutes), and complementary role to manual testing for edge/negative cases, despite challenges like incomplete OpenAPI specs and domain-specific inputs. Key lessons include: automated tools complement (but do not replace) manual tests; strict company policies may delay adoption; and academic terminology confuses practitioners.

The company has planned a broader rollout of RestTestGen and DeepREST across teams as strategic assets to identify unknown faults and check documentation conformance. Future research directions include new test generation algorithms to overcome current tool limitations, characterizing fault-triggering inputs to help users identify root causes, improving the comprehensibility of generated tests and reports, and applying test suite minimization to reduce redundancy while preserving fault-detection capability.

Acknowledgements

This work has received funding from SES and the Luxembourg National Research Fund under the Industrial Partnership Block Grant (IPBG), ref. IPBG19/14016225/INSTRUCT. The authors wish to thank Marco Kessler and the Open Orbits team for their availability in adopting the academic tools and for contributing to the empirical study.

References

[1] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, "REST APIs: A large-scale analysis of compliance with principles and best practices," in *International conference on web engineering*. Springer, 2016, pp. 21–39.

[2] A. Neumann, N. Laranjeiro, and J. Bernardino, "An analysis of public REST web service APIs," *IEEE Transactions on Services Computing*, vol. 14, no. 4, pp. 957–970, 2018.

[3] A. Premchand and A. Choudhry, "Open banking & APIs for transformation in banking," in *2018 international conference on communication, computing and internet of things (IC3IoT)*. IEEE, 2018, pp. 25–29.

[4] H. Sartaj, S. Ali, T. Yue, and K. Moberg, "Testing real-world healthcare IoT application: Experiences and lessons learned," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 2044–2049.

[5] K. Deepika, R. R. Sankar, and A. Satyanarayana, "RESTful API integrations in telecom billing system management," in *Proceedings of the International Conference on Cognitive and Intelligent Computing: ICCIC 2021, Volume 1*. Springer, 2022, pp. 323–329.

[6] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 748–758.

[7] 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, 2019.

[8] ——, "Automated black-and white-box testing of RESTful APIs with EvoMaster," *IEEE Software*, vol. 38, no. 3, pp. 72–78, 2020.

[9] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray, "Pythia: Grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations," *arXiv preprint arXiv:2005.11498*, 2020.

[10] S. Karlsson, A. Čaušević, and D. Sundmark, "QuickREST: Property-based test generation of OpenAPI-described RESTful APIs," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 131–141.

[11] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Black-box constraint-based testing of RESTful web APIs," in *International Conference on Service-Oriented Computing*. Springer, 2020, pp. 459–475.

[12] ——, "RESTest: automated black-box testing of RESTful web APIs," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 682–685.

[13] N. Laranjeiro, J. Agnelo, and J. Bernardino, "A black box tool for robustness testing of REST services," *IEEE Access*, vol. 9, pp. 24 738–24 754, 2021.

[14] J. C. Alonso, A. Martin-Lopez, S. Segura, J. M. Garcia, and A. Ruiz-Cortes, "ARTE: Automated generation of realistic test inputs for web APIs," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 348–363, 2022.

[15] D. Corradini, A. Zampieri, M. Pasqua, E. Viglianisi, M. Dallago, and M. Ceccato, "Automated black-box testing of nominal and error scenarios in RESTful APIs," *Software Testing, Verification and Reliability*, vol. 32, no. 5, p. e1808, 2022.

[16] H. Wu, L. Xu, X. Niu, and C. Nie, "Combinatorial testing of RESTful APIs," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 426–437.

[17] Z. Hatfield-Dodds and D. Dygalo, "Deriving semantics-aware fuzzers from web API schemas," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 345–346.

[18] Y. Liu, Y. Li, G. Deng, Y. Liu, R. Wan, R. Wu, D. Ji, S. Xu, and M. Bao, "Morest: Model-based RESTful API testing with execution feedback," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1406–1417.

[19] M. Kim, S. Sinha, and A. Orso, "Adaptive REST API testing with reinforcement learning," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 446–458.

[20] D. Corradini, Z. Montolli, M. Pasqua, and M. Ceccato, "DeepREST: Automated test case generation for REST APIs exploiting deep reinforcement learning," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1383–1394.

[21] T.-Q. Nguyen, N.-H. Cong, N.-M. Quach, H. D. Vo, and S. Nguyen, "Reinforcement learning-based REST API testing with multi-coverage," *arXiv preprint arXiv:2410.15399*, 2024.

[22] M. Kim, S. Sinha, and A. Orso, "LlamaRestTest: Effective REST API testing with small language models," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 465–488, 2025.

[23] T. Stennett, M. Kim, S. Sinha, and A. Orso, "AutoRestTest: A tool for automated REST API testing using LLMs and MARL," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2025, pp. 21–24.

[24] K. Zhang, C. Zhang, C. Wang, C. Zhang, Y. Wu, Z. Xing, Y. Liu, Q. Li, and X. Peng, "LogiAgent: Automated logical testing for REST systems with LLM-based multi-agents," *arXiv preprint arXiv:2503.15079*, 2025.

[25] Y. Liu, Y. Li, Y. Liu, R. Wan, R. Wu, and Q. Liu, "Morest: industry practice of automatic RESTful API testing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[26] M. Zhang, A. Arcuri, Y. Li, Y. Liu, and K. Xue, "White-box fuzzing RPC-based APIs with EvoMaster: An industrial case study," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–38, 2023.

[27] M. Zhang, A. Arcuri, P. Teng, K. Xue, and W. Wang, "Seeding and mocking in white-box fuzzing enterprise RPC APIs: An industrial case study," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2024–2034.

[28] M. Zhang, A. Arcuri, Y. Li, Y. Liu, K. Xue, Z. Wang, J. Huo, and W. Huang, "Fuzzing microservices: A series of user studies in industry on industrial systems with EvoMaster," *Science of Computer Programming*, p. 103322, 2025.

[29] A. Poth, O. Rrjolli, and A. Arcuri, "Technology adoption performance evaluation applied to testing industrial REST APIs," *Automated Software Engineering*, vol. 32, no. 1, p. 5, 2025.

[30] A. Arcuri, A. Poth, and O. Rrjolli, "Introducing black-box fuzz testing for REST APIs in industry: Challenges and solutions," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2025, pp. 382–393.

[31] X. Yu, L. Liu, X. Hu, J. Keung, X. Xia, and D. Lo, "Practitioners' expectations on automated test generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1618–1630.

[32] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[33] A. Lawi, B. L. Panggabean, and T. Yoshida, "Evaluating GraphQL and REST API services performance in a massive and intensive accessible information system," *Computers*, vol. 10, no. 11, p. 138, 2021.

[34] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, "RestTestGen: An extensible framework for automated black-box testing of RESTful APIs," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 504–508.

[35] O. Al-Debagy and P. Martinek, "A metrics framework for evaluating microservices architecture designs," *Journal of Web Engineering*, vol. 19, no. 3–4, pp. 341–370, 2020.

[36] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv *et al.*, "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025.

[37] A. Golmohammadi, M. Zhang, and A. Arcuri, "Testing RESTful APIs: A survey," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–41, 2023.

[38] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "Online testing of RESTful APIs: Promises and challenges," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 408–420.