RESTberus: Trace-Aware Benchmarking of Automated REST API Testing Tools Across Microservice Dependencies

John Toniutti
University of Udine – Italy
toniutti.john@spes.uniud.it

Massimiliano Baldo
IMT Lucca – Italy
massimiliano.baldo@imtlucca.it

Marino Miculan
University of Udine – Italy
marino.miculan@uniud.it

Vincenzo Riccio
University of Udine – Italy
vincenzo.riccio@uniud.it

arXiv:2611.04769v1 [cs.SE] 3 Nov 2026

***Abstract*—Existing REST API testing benchmarks focus on isolated services, overlooking failures arising from interactions with downstream dependencies. In microservice systems, response bodies and entry-point operations alone cannot reliably distinguish failures, compromising tool comparisons. We present RESTberus, a Kubernetes-based benchmarking platform for black-box REST API testing tools on deployed microservice systems. By complementing API-level evidence with distributed traces, RESTberus can distinguish failures manifesting in the target service from those manifesting in downstream dependencies. Across six tools on two microservice benchmarks, our trace-aware deduplication strategy achieves 87.1% accuracy, outperforming baselines from prior work by 10–13 percentage points. Our empirical study shows that all tools expose downstream failures, accounting for 11.7% of estimated unique failures on average, which isolated-service benchmarks cannot expose. On single-service benchmarks, RESTberus produces statistically equivalent results to RESTgym, demonstrating that integration-level assessments can preserve compatibility with existing tools and comparability with prior evaluations.**

***Index Terms*—REST APIs, Microservice Testing, Benchmarking Infrastructure, Distributed Tracing, Integration Testing**

I. Introduction

REST APIs are widely used to build the backends of modern web and cloud applications, especially in microservice architectures, where they allow clients to access backend functionality and services to communicate with each other [1]. Ensuring their reliability has motivated the development of several automated black-box testing tools, which generate requests from OpenAPI specifications to exercise API operations and expose failures. As these tools evolve, e.g., including deep learning approaches [2], reproducible benchmarking is essential to compare their effectiveness under controlled conditions.

However, existing benchmarking infrastructures focus on isolated services. In particular, RESTgym [3] provides a reproducible environment for comparing REST API testing tools, but evaluates individual services by observing HTTP responses at the API boundary and estimating unique failures through response-body similarity. This approach overlooks the challenges introduced by distributed microservice architectures.

In fact, a single API request within a microservice system may trigger calls to several downstream services before a response is returned. An HTTP 5xx response observed at the API boundary may therefore reflect a failure manifested either in the target service or in a downstream dependency. Moreover, intermediate services may return the same generic error response for different failures, while the same downstream failure may be exposed through different entry-point operations. Consequently, API-boundary observations alone cannot distinguish target-side from downstream failures, and grouping failures solely by response similarity or API operation may produce misleading estimates of unique failures.

Moving from isolated services to microservices requires more sophisticated orchestration and monitoring, but this technological leap should not come at the cost of comparability with established benchmarks.

MacroHive [4] successfully extends automated REST API testing to deployed microservice systems through dedicated test generation and monitoring of inter-service interactions. Although its approach is compared to testing tools, its empirical comparison does not apply the same internal observability and failure-analysis methodology to all compared tools.

This leaves an important gap: *how can we benchmark existing REST API testing tools on real microservice deployments, measure failures across service boundaries, and preserve comparability with established benchmarks?*

We address this gap with RESTberus, a Kubernetes-based benchmarking platform that evaluates existing black-box REST API testing tools without modifying their test-generation logic. RESTberus automatically deploys each target service with its downstream dependencies, isolates experimental runs, controls random seeds and interaction budgets, and collects API-boundary interactions, code coverage, and distributed traces. It introduces a trace-aware deduplication strategy to distinguish failures manifesting in the target service from those manifesting downstream. These capabilities are implemented within the benchmarking infrastructure, allowing existing approaches to benefit from richer evaluation without being redesigned.

We evaluate RESTberus using six testing tools on Train Ticket and PetClinic and nine single-service benchmarks from RESTgym. Our proposed deduplication strategy achieves 87.1% deduplication accuracy, outperforming existing strategies by up to 13.0 percentage points. It also achieves the highest observed accuracy (92.2%) on human-annotated failure pairs. All six testing tools expose downstream failures,

which account for 11.7% of their unique failures on average, demonstrating the importance of accounting for downstream failures. Finally, equivalence testing shows that RESTberus preserves the evaluation results of RESTgym on single-service benchmarks. Together, these findings demonstrate that benchmarking can move beyond API-boundary observations without sacrificing compatibility with existing tools or comparability with prior results.

The main contributions of this paper are:

- **Integration-level microservice benchmarking.** We introduce RESTberus to evaluate black-box REST API testing tools on deployed service chains, with controlled execution and integrated telemetry collection.
- **Trace-aware failure analysis.** We introduce attribution of target and downstream failure manifestations and a deduplication strategy that combines distributed traces with API-level evidence, improving upon existing approaches.
- **Empirical evidence and backward compatibility.** We evaluate six testing tools on distributed benchmarks, quantify the contribution of downstream failures, assess deduplication against LLM and human reference labels, and establish practical equivalence with RESTgym on single-service applications.

II. Background

A. REST APIs and Automated Black-Box Testing

REST is an architectural style for web services, commonly implemented over HTTP, in which clients manipulate server resources by issuing requests with a method, a resource path, and an optional payload [5]. Servers report outcomes with standardized status codes, where the 5xx class indicates server-side errors [6].

REST APIs are commonly documented with machine-readable specifications such as the OpenAPI Specification [7], which define operations, their inputs, and their expected responses. This contract enables black-box testing tools to exercise a service through its interface, without access to its source code [8], by deriving valid requests and sequences of HTTP interactions. Without access to internal program state, these tools rely on HTTP status codes as test oracles and commonly treat unexpected 5xx responses as failures.

B. Microservice Testing Levels

Testing a microservice-based application introduces challenges not present in monolithic architectures: network calls add latency, while independent deployment means an update to an API can cause compatibility issues [9], [10]. To manage this complexity, microservice testing is organized into multiple levels with distinct scopes and objectives [11]. In this paper, we focus on the transition from component level to integration testing (see Figure 1).

Component testing validates a single microservice through its exposed API while isolating external dependencies, such as downstream services and databases, typically using mocks or stubs [12]. This isolation confines failures to the microservice

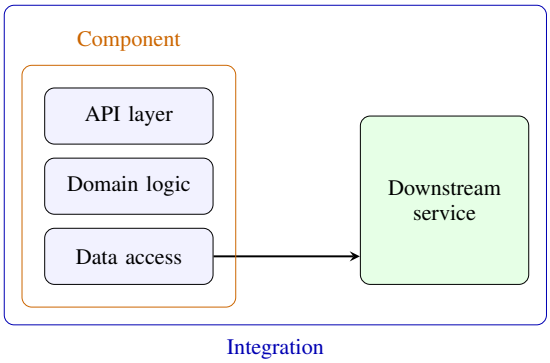


Fig. 1. Comparison between component and integration testing scopes.

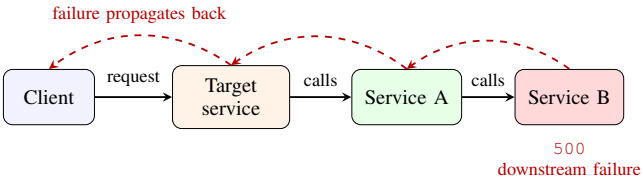


Fig. 2. Failure propagation across a chain of downstream microservice calls.

under test and makes executions more deterministic, as the behavior of external dependencies is explicitly controlled. However, this prevents tests from exercising actual inter-service interactions. Component testing validates a microservice only against its own assumptions about how its collaborators behave rather than their actual runtime behavior. Thus, failures that emerge only from interactions between services remain invisible at the component level.

Integration testing addresses this gap by deploying multiple interacting services and exercising them through their real interfaces [8], [11]. In an integration test, requests are sent to a target service while its downstream services remain deployed and active. An unexpected response from a downstream service may cause its caller to fail. As illustrated in Figure 2, a failure originating in a downstream service can propagate back through multiple callers [13], each of which may forward the response or replace it with a generic one. Therefore, different failing executions may produce indistinguishable responses at the API boundary, hiding important differences in their internal behavior.

C. Benchmarking Automated REST API Testing Tools: the RESTgym Platform

Benchmarking infrastructures enable a fair, empirical comparison of automated testing tools under controlled and reproducible conditions, and are widely used in software engineering, for instance for software fuzzing [14], [15], unit test generation [16], deep learning systems [17], and autonomous driving [18].

RESTgym [3] is the state-of-the-art benchmarking infrastructure for automated black-box REST API testing. To address the long-standing reproducibility challenges in REST

API testing research, it provides a Docker-based [19] environment pre-configured with a diverse suite of benchmark API services and testing tools. The orchestration engine automates testing sessions across API services, performing multiple repetitions to account for non-deterministic behavior of test generators. During each execution, RESTgym collects the HTTP interactions produced by the testing tool via mitmproxy [20], together with coverage information from the service under test. Following execution, RESTgym processes the collected data to derive three main performance metrics:

Operation Coverage measures the number of API operations declared in the OpenAPI specification that are successfully exercised with a 2xx response.

Code Coverage measures the line, branch, and method coverage of the service under test, collected with the JaCoCo library [21].

Unique Failures estimates the number of distinct failures exposed during testing. Any request resulting in an HTTP 5xx response is flagged as a potential failure. Because automated testing tools generate thousands of requests, the same failure may be observed repeatedly, making raw failure counts unrepresentative. To report unique failures, RESTgym implements a bucketing algorithm based on the Jaccard similarity index [22]. Response bodies are normalized to eliminate dynamic values (e.g., timestamps and IDs) and tokenized. Two failures are merged into the same bucket when their body similarity exceeds a threshold.

While this evaluation model is well suited to isolated services, applying it to distributed microservice systems exposes the following limitations:

- *Single-Service Assessment*: Existing benchmarking infrastructures, including RESTgym, evaluate tools on single-service APIs, and therefore cannot capture failures that arise only from interactions with real downstream services.
- *Insufficiency of Payload-Based Deduplication*: Bucketing 5xx responses by body similarity assumes that distinct failures produce distinguishable responses. In microservice systems, intermediate services or API gateways often turn downstream failures into the same generic error, so payload-only heuristics merge distinct failures and underestimate the number discovered.
- *Lack of Failure Attribution*: Single-service benchmarks assume that all observed failures originate in the service under test. In microservice systems, an entry-point failure may instead have manifested in a downstream dependency and propagated through the call chain. Without benchmark-side observability of these internal interactions, the evaluator cannot distinguish the two.
- *Random Seed Control*: Testing tools commonly rely on pseudo-random generators, so without systematic seed control by the orchestrator, executions are hard to reproduce and cross-tool comparisons are less controlled.

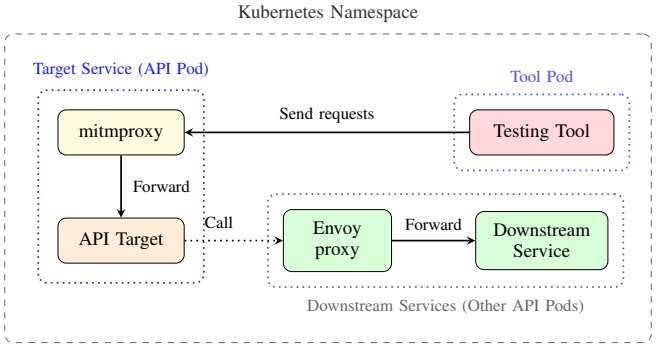


Fig. 3. Runtime topology of a RESTberus run within a Kubernetes namespace.

III. RESTberus

RESTberus is a Kubernetes-based [23] benchmarking infrastructure designed to evaluate automated REST API testing tools against microservice architectures. Kubernetes provides the orchestration mechanisms needed to deploy complete microservice dependency graphs, isolate concurrent experimental runs, and consistently manage the lifecycle and instrumentation of the services involved in each evaluation. Extending benchmarking REST API testing tools from isolated services to distributed systems, RESTberus automates the deployment of microservice architectures and collects runtime telemetry across service boundaries. This enables the evaluation of black-box testing tools in integration testing scenarios.

The design of RESTberus is guided by three main goals derived from the limitations discussed in Section II-C. First, *distributed deployment* enables testing tools to exercise real inter-service interactions rather than isolated services with mocked dependencies. Second, *internal observability* makes the execution of requests across service boundaries visible to RESTberus, enabling the attribution of observed failures to target or downstream services and trace-aware failure deduplication. Third, *controlled execution* isolates individual runs, standardizes the deployment and execution of testing tools and benchmark systems, and manages random seeds to support reproducible comparisons across experiments.

A. Architecture

RESTberus adopts a client-server architecture. An orchestrator engine coordinates the lifecycle of experiments and persists runtime telemetry. Users interact with the platform through a command-line interface to submit experiment configurations, monitor execution progress, and export evaluation reports. This decoupling allows the orchestrator to run either locally or directly inside a Kubernetes cluster, while benchmark systems and testing tools are instantiated as containerized workloads in Kubernetes. For each run, RESTberus deploys the target microservice together with its downstream services in an isolated namespace and launches the testing tool as a separate Kubernetes Job.

Figure 3 illustrates this runtime topology of a RESTberus run. Each component (the testing tool, the target service, and

each downstream service) executes inside its own *pod*, the basic Kubernetes unit grouping one or more tightly coupled containers. HTTP requests sent to the target service pass through a `mitmproxy` sidecar, which records request and response data observed at the API entry point. To complement these API-boundary observations, RESTberus instruments the deployed downstream services with Istio/Envoy proxies, which observe inter-service HTTP calls and generate distributed traces describing how each request propagates across the microservice chain. Test traffic reaches the target through mitmproxy without passing through Envoy, so client-to-target interactions are recorded at the API boundary, while distributed traces capture downstream inter-service calls. A trace is composed of individual *spans*, each representing one service-level operation or network call along that execution path. RESTberus also records code coverage measurements for the exercised services. The collected HTTP interactions, trace spans, and coverage measurements are stored in a local SQLite database [24] and later processed to compute aggregate evaluation metrics and reports.

B. Experiment Orchestration and Run Lifecycle

In RESTberus, evaluations are organized into *experiments* and *runs*. An experiment specifies a set of testing tools, a set of benchmark systems, and a number of repetitions. Each run is one execution of a single testing tool against a dedicated instance of a benchmark system. The orchestrator schedules these runs and manages their complete lifecycle.

The lifecycle of each evaluation run proceeds through three sequential phases: *deployment*, *execution*, and *cleanup*. During deployment, the orchestrator creates an ephemeral Kubernetes namespace dedicated to the specific run and deploys the benchmark system using Helm. The deployment includes the target API service alongside its downstream services and any required databases. Assigning each run to a separate namespace provides logical isolation between concurrent runs, preventing interference between their Kubernetes resources and service names. The namespace is also configured with the instrumentation required by RESTberus, including Istio/Envoy proxies for the downstream services and a `mitmproxy` sidecar for the target service.

Before executing the testing tool, RESTberus verifies that the deployed system is ready. It waits until all pod containers report a ready state and then probes the target API until it responds successfully. If either condition is not satisfied within a configured timeout, deployment is aborted, the namespace is marked as failed, and the run is scheduled for retry.

Once the deployment satisfies all readiness checks, the run enters the execution phase. The orchestrator launches the testing tool as a Kubernetes Job inside the same namespace, allowing it to communicate directly with the target service. To accommodate tools with different command-line interfaces and configuration formats without modifying tool source code, each tool is packaged in a container image with a standardized launch script. The orchestrator mounts the OpenAPI specification and input dictionaries required by the tool into the tool container, and passes runtime parameters (e.g., target URL and interaction budget) as environment variables. To control randomness in test generation, the orchestrator computes a distinct pseudo-random seed for each run based on the experiment seed and repetition index, injecting it into the tool container. Crucially, the tool pod is deliberately deployed without the Envoy sidecar normally injected by Istio, as shown in Figure 3. Since Envoy enforces strict HTTP validation, routing test traffic through it could reject malformed requests before they reach the target. Disabling this sidecar preserves the original traffic generated by the testing tool. During execution, background workers collect HTTP interactions, trace spans, and code coverage.

Following execution, the run concludes with the cleanup phase, regardless of whether testing completed successfully or aborted due to an error. Before tearing down the deployment, the orchestrator performs a final telemetry collection to capture observations from the last execution interval. It then deletes the namespace, releasing cluster resources for subsequent runs. All collected telemetry is stored for trace reconstruction and failure analysis, as described in Sections III-C and III-D.

C. Telemetry and Trace Reconstruction

To reconstruct how a test request propagates across microservices, RESTberus must associate each HTTP interaction recorded at the API entry point with the distributed trace describing its downstream calls. To establish this association, the `mitmproxy` sidecar injects a unique request identifier into each request entering the target service. This identifier is propagated across inter-service calls and recorded by Jaeger as an attribute of the corresponding trace spans. RESTberus uses this identifier to retrieve the spans associated with each HTTP interaction.

Once the spans for a request are retrieved, RESTberus reconstructs the observed inter-service call paths. Spans are ordered by their start times to establish their chronological order, while the parent link of each span is used to trace its call path back to the target service. These reconstructed paths provide the execution context for the failure analysis described in Section III-D.

D. Trace-Aware Failure Analysis

RESTberus analyzes observed failures through two complementary tasks. *Failure attribution* uses the reconstructed traces to classify each failure as a target failure, when it manifests in the target service, or a downstream failure, when it manifests in one of its dependencies. *Failure deduplication* groups equivalent failure occurrences into failure buckets to estimate the number of distinct failures.

Failure attribution identifies where a failure manifested in the observed trace. RESTberus examines the 5xx spans associated with a failing HTTP interaction and selects the latest-starting one, since downstream failures may propagate through multiple callers. A failure is attributed to a downstream service when this span belongs to a service other than the target, and to the target service otherwise. If no 5xx span is available

for an HTTP interaction, RESTberus classifies the failure as target-side by default. Relying on start time rather than parent links allows this heuristic to operate even when intermediate spans are missing, e.g., when a downstream call fails before the callee records its span or when a service times out leaving a gap in the captured execution path.

Failure deduplication decides whether two observed failures are the same or different. RESTberus adopts two matching criteria from prior work and introduces a third based on distributed traces. The *payload* criterion reproduces RESTgym's response body similarity bucketing described in Section II-C, with the same threshold of 0.7 [3], retrieved from its replication package. The *operation* criterion, adopted from MacroHive [4], matches failures associated with the same API operation, regardless of their response bodies. The *trace* criterion matches failures whose latest-starting 5xx spans belong to the same service, or whose reconstructed call paths are compatible. Compatibility requires the set of service-operation pairs from one path to be a subset of the other, which allows partial traces with missing spans to merge with more complete ones.

Each criterion has limitations. The payload criterion merges distinct failures masked by a generic error body, as discussed in Section II-C. The operation criterion merges distinct failures reached through the same operation, as happens when an operation calls several downstream services. The proposed trace criterion requires spans, which may be unavailable for every failure, since a failure raised by the target service without any downstream call produces no span.

Based on these criteria, RESTberus implements five deduplication strategies. The `payload_only` and `operation_only` strategies apply their respective criteria independently and serve as baselines. `trace_payload` merges two traced failures only when both the trace and payload criteria hold, and applies the payload criterion alone to untraced failures. Similarly, `trace_operation` combines trace compatibility with operation matching for traced failures and applies operation matching to untraced failures. The `hybrid` strategy handles traced failures as `trace_payload`, but groups untraced failures by API operation. This hybrid design exploits complementary evidence depending on trace availability: for traced failures, payload similarity helps distinguish different failure manifestations along compatible execution paths; for untraced failures, operation-based grouping avoids merging failures from different API operations solely because they produce the same generic response. The three trace-aware strategies never merge traced and untraced failures, since execution-trace evidence is available for only one failure and cannot support a direct comparison of their internal behavior.

IV. Empirical Setup

A. Research Questions

RQ1 (Failure Deduplication): *To what extent do trace-aware strategies improve failure deduplication accuracy in microservice systems?*

Existing approaches estimate unique failures using different criteria: RESTgym groups HTTP 5xx responses by body similarity, while MacroHive counts distinct failing API operations. In microservice architectures, however, different downstream failures may produce indistinguishable responses or occur through the same API operation, causing these deduplication criteria to merge distinct failures. RQ1 investigates whether trace-aware strategies improve deduplication accuracy over these existing approaches. We evaluate the strategies against human-calibrated LLM reference labels for sampled failure pairs, and assess the effectiveness of existing testing tools according to the best-performing strategy.

RQ2 (Downstream Failure Attribution): *What proportion of unique failures observed during REST API testing are attributed to downstream dependencies rather than to the target service?*

In microservice architectures, an HTTP 5xx response observed at the entry point may reflect a failure manifested in a downstream microservice rather than in the target service. Since entry-point observations alone cannot distinguish these cases, RQ2 uses RESTberus trace-based attribution to quantify the prevalence of unique failures attributed to downstream services.

RQ3 (Platform Equivalence): *Does RESTberus produce equivalent evaluation results to RESTgym when testing single-service applications?*

Although RESTberus extends benchmarking to microservice systems, its Kubernetes-based architecture introduces differences in orchestration and networking. To assess whether these changes influence evaluation outcomes, RQ3 compares RESTberus and RESTgym using the same testing tools and a common subset of single-service benchmarks under the same experimental conditions. We assess whether differences in the resulting evaluation metrics remain within predefined margins of practical equivalence.

B. Testing Tools

To evaluate RESTberus across representative testing methodologies, we considered the automated black-box REST API testing tools already assessed in prior benchmarking studies [3], [25]. We selected tools from this pool according to the following criteria:

1) *OAS 3.0 Compatibility*: Tools must support OpenAPI 3.0 specifications and generate tests for the benchmark APIs;
2) *Availability*: Tools must have publicly available implementations;
3) *Technological Diversity*: The tool set should cover different test generation approaches to provide a comprehensive assessment of REST API testing techniques.

Based on these criteria, we included six tools:

- **DeepREST** [2]: Uses reinforcement learning to handle implicit operation dependencies and parameter constraints;

- **MoREST** [26]: Constructs a graph from the OpenAPI specification to model operations, data schemas, and producer-consumer relationships;
- **RESTest** [27]: Applies constraint-based testing to handle inter-parameter dependencies, including mutually exclusive or conditionally required parameters;
- **RESTler** [28]: Uses grammar-based fuzzing, compiling OpenAPI specifications in request generation grammars;
- **RestTestGen** [29]: Constructs a graph from the OpenAPI specification based on input-output dependencies between API operations to guide test generation;
- **Schemathesis** [30]: Applies property-based testing to generate inputs from OpenAPI specifications.

All six tools exercise the target service through its REST API, using its OpenAPI specification to guide test generation, and observe the responses at that boundary. Since internal service interactions and distributed traces are not exposed to the tools, their test generation logic remains unchanged when moving from isolated services to microservice deployments.

C. Experimental Objects

To evaluate RESTberus across both single-service baselines and distributed environments, we selected three benchmark suites:

Train Ticket [31]: A train booking system composed of 41 microservices, recognized by a recent systematic review as the de-facto standard benchmark for large microservice architectures [10], and used in prior work on integration testing and fault localization [11], [32]. Its distributed architecture and inter-service dependencies make it suitable for evaluating how REST API testing tools exercise multi-service interactions and expose downstream failures. Following the integration testing configurations by Gregor et al. [11], we organized Train Ticket into 19 test subsets, each comprising one target service and all the downstream services it directly or transitively invokes. This allows tools to exercise real inter-service interactions without deploying the complete application for every experiment. To support controlled testing and failure measurement, we applied the following adaptations:

- *Circuit-Breaker Fallback Removal*: In the original Train Ticket implementation, operations are instrumented with Hystrix [33] circuit breakers. When a downstream call fails, the circuit breaker intercepts the exception and returns a synthetic response (e.g., an empty payload or an HTTP 200 OK response). Although these fallbacks improve service resilience in production, they may interfere with the failure oracle of automated black-box REST API testing tools, which detect failures through HTTP 5xx responses. A downstream failure converted into a successful response is never reported, since the tools only see the response returned at the API boundary. Thus, we disabled these fallback handlers so that otherwise unhandled downstream failures can propagate back to the caller as HTTP 5xx responses and remain observable.
- *Timeout Reduction*: We reduced the socket timeout of inter-service HTTP calls from 5 seconds to 500 ms, so that boundary payloads that make downstream services hang do not stall the testing tool and lower test throughput.
- *Distributed Tracing*: We integrated Spring Cloud Sleuth [34] to enable trace collection across services, by propagating W3C trace context and the injected request identifier across outgoing HTTP calls.
- *OpenAPI Specification Migration*: We migrated legacy Swagger 2.0 specifications to OpenAPI 3.0 using springdoc [35] and validated all resulting specifications with Spectral [36] and manual review.
- *Coverage and Telemetry Instrumentation*: We configured services to retain exception details in HTTP error responses, preserving diagnostic information for response-based failure analysis. We also packaged each container image with the JaCoCo agent to collect runtime code coverage.

PetClinic [37]: A distributed implementation of the Spring PetClinic application composed of 7 microservices. As a smaller distributed benchmark, it complements Train Ticket and allows us to assess trace-aware failure analysis across different microservice applications and dependency structures. We applied the following adaptations:

- *Removal of External Dependencies and Static Routes:* We removed dependencies on services outside the benchmark deployment and excluded static-resource routes, so that observed failures come from REST operations within the deployed system.
- *OpenAPI 3.0 Specification Generation:* We integrated springdoc [35] to generate OpenAPI 3.0 specifications for the gateway-exposed REST routes.
- *Distributed Tracing and Telemetry Instrumentation:* Unlike Train Ticket, PetClinic natively supports distributed tracing. We configured it to sample 100% of traces. As with Train Ticket, we configured error handling to retain exception details in HTTP error responses and instrumented the services with JaCoCo to collect runtime code coverage.

Single-Service Benchmarks: To address RQ3, which evaluates whether RESTberus reproduces RESTgym results on single-service applications, we included 9 of the 17 services available in the RESTgym repository [38]. We selected the services on which all six evaluated tools could execute to completion on both platforms, ensuring a balanced paired design across all evaluation metrics.

D. Experimental Protocol

To ensure fair comparison across tools with different generation speeds, we bounded each run by an interaction budget of 10,000 HTTP requests rather than a wall-clock timeout. A preliminary convergence analysis showed that growth slows well before this limit. Code coverage and unique failures reached 95% of their final values within 5,500 and 6,000 requests, respectively, while all metrics reached 99% by 9,500 requests. Each tool-service combination was executed ten times, with a

distinct random seed to account for non-deterministic exploration [39] while supporting reproducibility.

For RQ3, we evaluated the same six tools on nine single-service benchmarks using both RESTberus and RESTgym, with ten repetitions and the same 10,000-request budget on each platform. For each tool-service combination, we averaged the ten repetitions separately for each platform, obtaining a matched pair of measurements for the statistical analyses.

RESTberus performs an eligibility check to verify run validity, thus ensuring data quality: a run is deemed valid only if it completed its full interaction budget and recorded coverage samples throughout execution. Runs that terminate prematurely or stall are automatically retried with the same random seed and parameters. Ten runs required a retry due to temporary cluster resource exhaustion.

E. Evaluation Metrics

Runs are processed to compute metrics to answer each research question. We consider HTTP 5xx responses observed at the API entry point as potential failures, consistently with RESTgym. To estimate the number of *Unique HTTP 5xx Failures*, we group these observations into buckets using the deduplication strategies described in Section III-D. Each bucket represents an estimated distinct failure manifestation.

In RQ1, we assess deduplication against reference labels using *Accuracy* (the proportion of correctly classified failure pairs), *Precision* (the proportion of predicted duplicates that are actual duplicates), and the *Matthews Correlation Coefficient (MCC)* [40] (the correlation between predicted and reference binary labels, i.e., DUPLICATE or DISTINCT). MCC accounts for all four outcomes of binary classification and is informative under class imbalance, ranging from -1 (perfect disagreement) through 0 (no correlation) to $+1$ (perfect agreement). To evaluate testing tool effectiveness, we measure *Operation Coverage*, *Line Coverage*, and *Unique HTTP 5xx Failures*.

In RQ2, following Section III-D, we partition unique failures into *Target Failures* and *Downstream Failures*. We report both counts, their sum (*Total Failures*), and the *Downstream Failure Ratio* (i.e., the percentage of unique failures attributed to downstream dependencies).

In RQ3, we evaluate five metrics across single-service benchmarks: *Operation Coverage*, *Line Coverage*, *Branch Coverage*, *Method Coverage*, and *Unique HTTP 5xx Failures*.

F. Experimental Procedure

All experiments were executed on an isolated Linux virtual machine equipped with 64 vCPUs of an Intel Xeon 6342 processor and 96 GB of RAM. The Kubernetes cluster environment was managed using k3s [41]. Executing the full experimental suite across Train Ticket, PetClinic, and the single-service benchmarks required approximately 10 days.

To establish the reference labels for RQ1, we collected an evaluation dataset of failure pairs using a two-tier stratified sampling strategy across all experimental runs. For each run, we sampled 15 failure pairs uniformly at random to capture general testing behavior, and 5 additional pairs where both failures carried distributed traces, ensuring representation of downstream microservice interactions. As some runs did not trigger enough failures, fewer pairs were sampled in those runs, yielding $N = 21{,}997$ pairs in total. Each pair was classified as DUPLICATE or DISTINCT by a multi-model LLM judge ensemble (Qwen 3.8 27B and Gemma 4 31B, with DeepSeek V4 Flash resolving ties). The judges used entry-point HTTP responses, trace call sequences, and container logs, with four few-shot demonstration examples and temperature set to zero to minimize sampling variability. To confirm the reliability of the LLM reference labels, we manually annotated a calibration sample of 346 failure pairs and measured agreement using Cohen's κ [42]. The LLM consensus achieved 87.9% agreement with these annotations (Cohen's $\kappa = 0.758$), indicating substantial agreement [43]. The two primary judges agreed on 89.0% of pairs ($\kappa = 0.780$), with the arbiter resolving the remaining 11.0% of disagreements.

We evaluated the five deduplication strategies against both the LLM reference labels and the human annotations, using McNemar's test [44] to assess differences in classification accuracy. We then compared testing tool effectiveness across Train Ticket and PetClinic using the best-performing deduplication strategy.

In RQ2, we investigated failure attribution by categorizing unique failures into target and downstream failures according to the attribution criterion of Section III-D. In particular, we computed unique-failure counts using the best-performing deduplication strategy identified in RQ1. We compared tools on target, downstream, and total unique failure counts across benchmark services and computed the proportion of failures attributed to downstream dependencies.

In RQ3, we assessed platform equivalence between RESTberus and RESTgym across the nine single-service benchmarks. To ensure a like-for-like comparison, unique failures were computed using the `payload_only` strategy on both platforms, reproducing the response-based bucketing adopted by RESTgym. In fact, as the single-service benchmarks are not traced, `trace_payload` falls back to the same payload comparison and would yield identical counts, whereas `trace_operation` and `hybrid` fall back to grouping failures by operation instead of by response body and would yield different counts by design. We assessed systematic differences in the five evaluation metrics and tested practical equivalence against predefined margins of ± 1 operation, ± 1 percentage point for code coverage, and ± 1 unique failure. Equivalence was established when the Hodges–Lehmann 90% confidence interval [45] for a paired difference lay entirely within the corresponding margin, according to equivalence testing principles [46].

Comparisons of testing-tool effectiveness in RQ1 and RQ2, as well as comparisons between platforms in RQ3, used two-sided paired Wilcoxon signed-rank tests [47] on the means of the ten repetitions, with a significance threshold of $\alpha = 0.05$.

TABLE I
RQ1: FAILURE DEDUPLICATION PERFORMANCE ACROSS EVALUATION SETS. BEST RESULTS IN BOLD; UNDERLINED STRATEGIES ARE NOT STATISTICALLY DIFFERENT FROM THE BEST.

Heuristic	Accuracy (%)	Precision (%)	MCC
Traced (LLM consensus, $N = 4{,}725$)			
`payload_only`	82.0	80.6	0.633
`operation_only`	64.6	64.8	0.264
`trace_payload`	**91.0**	**91.7**	**0.818**
`trace_operation`	77.8	76.6	0.547
`hybrid`	**91.0**	**91.7**	**0.818**
All Pairs (LLM consensus, $N = 21{,}997$)			
`payload_only`	74.1	69.5	0.498
`operation_only`	77.4	75.1	0.550
`trace_payload`	78.5	74.0	0.582
`trace_operation`	83.1	84.4	0.662
`hybrid`	**87.1**	**90.2**	**0.745**
Human calibration sample ($N = 346$)			
`payload_only`	77.5	70.0	0.584
`operation_only`	79.8	74.6	0.606
`trace_payload`	80.6	73.4	0.640
`trace_operation`	90.5	91.4	0.809
`hybrid`	**92.2**	**94.3**	**0.845**

V. RESULTS

A. RQ1 (Failure Deduplication)

Table I reports the performance of the five deduplication strategies. On traced pairs, `hybrid` and `trace_payload` achieve identical results, since they apply the same matching criteria when traces are available. Both outperform `payload_only` (91.0% vs. 82.0% accuracy), because traces separate distinct downstream failures that gateways mask behind the same error body. `trace_operation` drops to 77.8% accuracy on these pairs. In more than half of the traced pairs, both failures manifest in the target after successful downstream calls, so the trace adds no evidence and only the response body separates distinct failures of the same operation, e.g., a timeout from a null pointer exception. When both failures manifest in the same downstream service, the entry operation is also a poor proxy, as the same failure is often reached through different operations. These limitations also explain why `operation_only` achieves the lowest accuracy on these pairs (64.6%). Across all sampled pairs, `hybrid` achieves the best results (87.1% accuracy, 0.745 MCC). McNemar's test confirms that, on these pairs, `hybrid` differs significantly from every other strategy ($p < 0.001$).

To mitigate potential evaluation bias arising from using LLM consensus, we evaluated all five strategies on the manually annotated calibration sample. On this manual benchmark, `hybrid` again achieves the highest accuracy (92.2%) and MCC (0.845). `hybrid` significantly outperforms `payload_only`, `operation_only`, and `trace_payload` ($p < 0.001$), while `trace_operation` achieves comparably high accuracy, with no statistically significant difference from `hybrid` ($p = 0.109$) and only 10 disagreements across 346 pairs. Thus, `hybrid` is the most accurate deduplication strategy whether the reference labels come from the LLM judges or from manual annotation.

TABLE II
RQ1: PER-TOOL EFFECTIVENESS ON TRAIN TICKET AND PETCLINIC, AVERAGED ACROSS SERVICES. BEST RESULTS IN BOLD; UNDERLINED VALUES ARE NOT STATISTICALLY DIFFERENT FROM THE BEST.

Tool	Operation Coverage	Line Coverage (%)	Unique 5xx failures
DeepREST	**11.59**	**55.1**	9.80
MoREST	5.83	45.5	5.45
RESTest	7.98	51.4	9.21
RESTler	7.09	50.0	7.74
RestTestGen	10.21	52.8	8.98
Schemathesis	8.15	49.4	**14.56**
Average	8.48	50.7	9.29

Therefore, we use `hybrid` to assess the unique failures discovered by each tool, alongside coverage metrics. Table II reports these metrics averaged across repetitions and benchmark services in Train Ticket and PetClinic.

DeepREST achieves the highest operation coverage (11.59), and ties statistically with RestTestGen for highest line coverage (55.1% vs. 52.8%, $p = 0.083$). Schemathesis discovers the highest number of unique failures (14.56 per run), significantly outperforming all other tools ($p < 0.001$).

> RQ1: The proposed `hybrid` strategy outperforms all alternative deduplication strategies, achieving 87.1% accuracy and 90.2% precision across an evaluation sample including both traced and untraced failures, with improvements of up to 13.0 and 20.7 percentage points, respectively. Its effectiveness is further supported by the highest observed accuracy (92.2%) on human-annotated failure pairs.

B. RQ2 (Downstream Failure Attribution)

To answer RQ2, we investigate how frequently testing tools trigger failures beyond the target service. Using the latest-starting 5xx trace span, we classify failures as either local to the target service or manifesting in a downstream service.

Table III reports the cumulative number of unique target and downstream failures detected by each tool across Train Ticket and PetClinic, averaged over repetitions per service and summed across services.

All six tools expose downstream failures, which account for an average of 11.7% of their unique failures, ranging from 8.1% to 14.7% across tools. DeepREST detects the most downstream failures (28.8), although its differences from RESTest, RestTestGen, and Schemathesis are not statistically significant. Conversely, Schemathesis detects the most target-service failures (267.6), significantly outperforming all other tools. These results show that tool rankings depend on where failures manifest, highlighting information that cannot be obtained from API-boundary responses alone.

RQ2: Downstream failures account for an average of 11.7% of unique failures across all six testing tools. By distinguishing these failures from target-service failures, RESTberus trace-based attribution provides an additional dimension for assessing testing-tool effectiveness.

C. RQ3 (Platform Equivalence)

To answer RQ3, we investigate whether RESTberus reproduces the evaluation results of RESTgym on single-service benchmarks, despite differences in orchestration and networking. We compare the platforms across five metrics: operation coverage, line, branch, and method coverage, and unique HTTP 5xx failures. We report unique failures using `payload_only` on both platforms, to reproduce RESTgym's original deduplication criterion. For each tool-service combination, the mean over ten repetitions on each platform forms a matched pair for statistical analysis.

Table IV reports the aggregate mean performance of both platforms. To compare the platforms, we evaluate two-sided Wilcoxon signed-rank tests to check for systematic bias, and Hodges-Lehmann 90% confidence intervals against predefined practical equivalence margins. These margins represent the smallest differences of practical interest: ± 1 operation (the minimal discrete unit), ± 1 percentage point for code coverage metrics (comparable to the variation between repeated runs of the same tool), and ± 1 for unique failures.

For all five metrics, the two-sided paired Wilcoxon signed-rank tests confirm no statistically significant differences between platforms ($p \geq 0.21$). Their Hodges-Lehmann 90% confidence intervals also lie entirely within the equivalence margins. These results establish practical equivalence between the two platforms under the evaluated conditions.

RQ3: Across the considered single-service benchmarks, RESTberus produces statistically equivalent evaluation results to RESTgym across all five metrics, with confidence intervals entirely within the predefined equivalence margins. This demonstrates that our extended benchmarking infrastructure preserves comparable tool assessments on single-service applications.

TABLE III
RQ2: TARGET AND DOWNSTREAM FAILURE ATTRIBUTION ON TRAIN TICKET AND PETCLINIC. BEST RESULTS IN BOLD; UNDERLINED VALUES ARE NOT STATISTICALLY DIFFERENT FROM THE BEST.

Tool	Target	Downstream	Total	Downstream ratio (%)
DeepREST	167.2	**28.8**	196.0	14.7
MoREST	96.8	12.1	108.9	11.1
RESTest	158.2	<u>26.1</u>	184.3	14.2
RESTler	139.8	14.9	154.7	9.6
RestTestGen	154.7	<u>25.0</u>	179.7	13.9
Schemathesis	**267.6**	<u>23.7</u>	**291.3**	8.1
Average	164.0	21.8	185.8	11.7

TABLE IV
RQ3: COMPARISON RESTBERUS VS RESTGYM ACROSS BENCHMARK SERVICES. UNDERLINED VALUES DENOTE STATISTICAL EQUIVALENCE.

Metric	RESTgym	RESTberus
Operation coverage	11.12	<u>11.51</u>
Line coverage (%)	56.2	<u>56.7</u>
Branch coverage (%)	23.9	<u>23.8</u>
Method coverage (%)	59.4	<u>59.9</u>
Unique 5xx failures	7.42	<u>7.14</u>

D. Threats to Validity

Internal Validity concerns the extent to which the observed effects are caused by experimental factors rather than uncontrolled variables. To mitigate these threats, all tools ran under identical conditions with a request budget instead of a time limit, limiting the effect of cluster throughput. Moreover, we isolated each run in a dedicated Kubernetes namespace and performed ten repetitions with controlled random seeds. Incomplete or stalled runs were retried with identical parameters.

Construct Validity concerns whether the adopted evaluation measures accurately capture the intended concepts. We consider HTTP 5xx responses as potential failures and estimate failure uniqueness through deduplication. However, determining unique failures is challenging in black-box REST API testing, and manually inspecting tens of thousands of failure pairs is infeasible. To assess automated deduplication accuracy, we used reference labels from a multi-model LLM judge ensemble, mitigating potential labeling errors through model diversity, zero-temperature prompting, and validation against 346 manually annotated pairs ($\kappa = 0.758$). For failure attribution, RESTberus requires trace evidence to classify a failure as downstream; otherwise, it conservatively attributes it to the target service. This avoids unsupported downstream attributions, although missing or incomplete spans may cause some downstream manifestations to be overlooked.

External Validity concerns the generalizability of our findings. We selected six testing tools representing distinct generation paradigms across two distributed benchmark systems. Both benchmarks use Java and Spring, which limits generalizability to polyglot architectures or alternative communication protocols, but they differ in size and dependency structure. Moreover, circuit-breaker fallbacks in Train Ticket were disabled so that downstream failures remain observable at the API boundary. Compared with a production deployment that retains such resilience mechanisms, more downstream failures are visible to the tools, so our downstream-failure ratios characterize what tools can expose in hardened systems rather than replicate production conditions.

Conclusion Validity concerns the degree to which our statistical analysis supports our conclusions. To account for stochastic test generation, we repeated each execution ten times [39] and used paired non-parametric tests. For RQ1, we compared deduplication strategies using McNemar's test and corroborated LLM-based results with human annotations. For RQ3, we did not infer equivalence from non-significant

differences alone, but additionally required the 90% confidence intervals of paired differences to lie entirely within predefined practical equivalence margins.

VI. Lessons Learned and Actionable Insights

Benchmark complete service chains, not isolated APIs. Benchmarking testing tools exclusively on isolated services overlooks failure modes arising from interactions with real downstream dependencies, e.g., cascading timeouts or downstream crashes. Across our distributed benchmarks, all six testing tools exposed downstream failure manifestations, accounting for 11.7% of unique failures on average. Benchmarking campaigns should complement single-service evaluations with deployed microservice systems and assess failures across service boundaries.

Look beyond response bodies and API operations to distinguish failures. Similar HTTP error responses may correspond to different failures, while the same downstream failure may be triggered through different API operations. Our `hybrid` deduplication strategy combines traces and response bodies when traces are available, and operation-based grouping otherwise, achieving 87.1% deduplication accuracy compared with 74.1% for the payload-only strategy adopted by RESTgym and 77.4% for the operation-only strategy adopted by MacroHive. Benchmarking infrastructures must leverage inter-service execution evidence, such as distributed traces, to observe how requests propagate, distinguish target-service from downstream failures, and deduplicate failures accurately.

Modernize benchmarking infrastructure without breaking existing tools. Supporting distributed applications requires more sophisticated deployment and observation mechanisms, but these changes should not alter the evaluation of existing tools on established benchmarks. RESTberus runs existing REST API testing tools without modification and achieves statistically equivalent results to RESTgym. Benchmarking platforms should preserve tool compatibility and established metrics, allowing researchers to build on existing tools and empirical results while extending their evaluation to more realistic distributed environments.

VII. Related Work

A. Benchmarking Automated REST API Testing

REST API relevance has motivated several automated test generators, whose effectiveness is usually assessed through API operation coverage, code coverage, and observed failures [1]. RESTgym [3] facilitates reproducible comparisons of black-box testing tools by providing containerized benchmark applications, standardized execution, and common evaluation metrics. However, its benchmarks target individual services, and its failure deduplication relies only on response-body similarity. Our approach extends controlled tool comparisons to deployed microservice dependencies and incorporates distributed traces to improve failure analysis, while preserving comparable results on single-service benchmarks.

Other work offers experimental resources for evaluating REST API testing. Gregor et al. [11] provide manually developed component- and integration-level test suites and a fault dataset for Train Ticket. Defects4REST [48] offers a benchmark of real-world REST API defects, enabling effectiveness studies against known defects. WFC/WFD [49] provides reusable API subjects and common specifications for authentication, fault reporting, and experimental comparisons. These contributions are complementary to RESTberus, which focuses on reproducibly executing test generators against deployed microservice systems and measuring their effectiveness through consistent, trace-aware analysis.

B. Microservice Testing and Runtime Evidence

MacroHive [4] combines automated test generation with service monitoring and causal analysis to expose failures across microservice dependencies, including internal failures masked at the API boundary. It provides an asymmetrical comparison with existing tools, as it collects internal failure information for MacroHive while evaluating the other tools through API-boundary metrics. Unlike MacroHive, RESTberus applies the same trace-aware failure analysis procedure to every evaluated tool, enabling comparisons of target and downstream failures without changing test generation logic. Our approach also differs in scope: MacroHive can observe internal failures independently of the entry-point response, whereas RESTberus analyzes failures associated with observed HTTP 5xx responses.

Other approaches exploit runtime observations to improve microservice testing. MicroFuzz [50] combines distributed tracing with service mocking to support efficient fuzzing, while MISH [51] learns from microservice logs to guide EvoMaster's test generation. Zhang et al. [52] also demonstrate the application of EvoMaster to industrial microservice systems. Instead of introducing another generation technique, RESTberus evaluates existing black-box generators under common experimental conditions.

Distributed traces have also been used for other testing purposes. Abdelfattah et al. [53] use traces to measure microservice endpoint coverage, while Rios et al. [32] use them for fault localization. Instead, RESTberus uses traces to distinguish failures manifesting in target and downstream services and to improve failure deduplication, enabling more informative comparisons of REST API testing tools.

VIII. Conclusions and Future Work

We presented RESTberus, a Kubernetes-based benchmarking platform for evaluating black-box REST API testing tools on microservice systems. RESTberus introduces a failure analysis strategy that leverages distributed traces to distinguish target-service failures from downstream failures and to deduplicate them. Across six tools, our novel deduplication strategy achieved 87.1% deduplication accuracy, compared with 74.1% and 77.4% for payload-only and operation-only strategies proposed in existing work. Our assessment considered failures

wherever they manifest: all testing tools exposed both target-side and downstream failures, with the latter accounting for 11.7% of unique failures on average. Schemathesis discovered the most unique failures overall. Our study demonstrates that benchmarking can move beyond API-boundary observations without sacrificing compatibility with existing tools or comparability with prior results.

Future work will extend RESTberus to support real-world polyglot systems, including Go and Python. We also plan to expose runtime traces as feedback for microservice-aware test generation.

REFERENCES

[1] 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, 2024.

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

[3] D. Corradini, M. Pasqua, and M. Ceccato, "RESTgym: A flexible infrastructure for empirical assessment of automated REST API testing tools," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2025, pp. 757–761.

[4] L. Giamattei, A. Guerriero, R. Pietrantuono, and S. Russo, "Automated functional and robustness testing of microservice architectures," *Journal of Systems and Software*, vol. 207, p. 111857, 2024.

[5] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[6] R. T. Fielding, M. Nottingham, and J. Reschke, "HTTP Semantics," RFC 9110, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9110

[7] OpenAPI Initiative, "OpenAPI Specification," https://spec.openapis.org/oas/latest.html.

[8] H. Washizaki, Ed., *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), Version 4.0*. IEEE Computer Society, 2024.

[9] L. Gregor, A. Hentschel, L. Kastner, and A. Pretschner, "A taxonomy of integration-relevant faults for microservice testing," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2025, pp. 138–149.

[10] F. Ponce, R. Verdecchia, B. Miranda, and J. Soldani, "Microservices testing: A systematic literature review," *Information and Software Technology*, vol. 188, p. 107870, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584925002095

[11] L. Gregor, M. Skalski, and A. Pretschner, "Benchmarking component and integration testing in microservices: Test suites and fault analysis on TrainTicket," in *2025 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2025, pp. 39–50.

[12] T. Clemson, "Testing strategies in a microservice architecture," https://martinfowler.com/articles/microservice-testing/, November 2014.

[13] J. Soldani, S. Forti, L. Roveroni, and A. Brogi, "Explaining microservices' cascading failures from their logs," *Software: Practice and Experience*, vol. 55, no. 5, pp. 809–828, 2025. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3400

[14] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "FuzzBench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1393–1403. [Online]. Available: https://doi.org/10.1145/3468264.3473932

[15] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang, "UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2777–2794. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei

[16] X. Devroey, A. Gambi, J. P. Galeotti, R. Just, F. Kifetew, A. Panichella, and S. Panichella, "JUGE: An infrastructure for benchmarking Java unit test generators," *Software Testing, Verification and Reliability*, p. e1838, 2023.

[17] M. Maryam, M. Biagiola, A. Stocco, and V. Riccio, "GIFTbench: Generative image fuzz testing benchmark," *Science of Computer Programming*, vol. 253, p. 103502, 2026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642326000687

[18] M. Biagiola, S. Klikovits, J. Peltomäki, and V. Riccio, "SBFT tool competition 2023 - cyber-physical systems track," in *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE, 2023, pp. 45–48.

[19] Docker, Inc., "Docker," https://www.docker.com/.

[20] mitmproxy Project, "mitmproxy," https://www.mitmproxy.org/.

[21] JaCoCo authors, "JaCoCo Java code coverage library," https://www.eclemma.org/jacoco/.

[22] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et des Jura," *Bull Soc Vaudoise Sci Nat*, vol. 37, no. 142, pp. 547–579, 1901.

[23] The Kubernetes Authors, "Kubernetes," https://kubernetes.io/.

[24] SQLite Consortium, "SQLite," https://www.sqlite.org/.

[25] M. Pasqua, D. Corradini, S. Mari, and M. Ceccato, "SBFT tool competition 2026: REST league," in *Proceedings of the 19th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, ser. SBFT '26. New York, NY, USA: Association for Computing Machinery, 2026, p. 47–51. [Online]. Available: https://doi.org/10.1145/3786155.3795704

[26] 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*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1406–1417. [Online]. Available: https://doi.org/10.1145/3510003.3510133

[27] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: automated black-box testing of RESTful web APIs," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 682–685. [Online]. Available: https://doi.org/10.1145/3460319.3469082

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

[29] 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)*, 2022, pp. 504–508.

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

[31] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2021.

[32] J. Rios, S. Jha, and L. Shwartz, "Localizing and explaining faults in microservices using distributed tracing," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, 2022, pp. 489–499.

[33] Netflix, "Hystrix," https://github.com/Netflix/Hystrix.

[34] The Spring Cloud Team, "Spring Cloud Sleuth," https://spring.io/projects/spring-cloud-sleuth.

[35] springdoc authors, "springdoc-OpenAPI," https://springdoc.org/.

[36] Stoplight, "Spectral," https://stoplight.io/open-source/spectral.

[37] Spring PetClinic authors, "Spring PetClinic microservices," https://github.com/spring-petclinic/spring-petclinic-microservices.

[38] RESTgym authors, "RESTgym repository," https://github.com/restgym/restgym, commit 9a9b1e9, June 2026.

[39] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 1–10.

[40] D. Chicco and G. Jurman, "The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation," *BMC genomics*, vol. 21, no. 1, pp. 6–13, 2020.

[41] Rancher / SUSE, "k3s: Lightweight Kubernetes," https://k3s.io/.

[42] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[43] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977.
[44] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, no. 2, pp. 153–157, 1947.
[45] J. L. Hodges and E. L. Lehmann, "Estimates of location based on rank tests," *The Annals of Mathematical Statistics*, vol. 34, no. 2, pp. 598–611, 1963.
[46] S. Wellek, *Testing Statistical Hypotheses of Equivalence and Noninferiority*, 2nd ed. Chapman and Hall/CRC, 2010.
[47] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.
[48] R. Mehta, P. Katkhede, and M. Motwani, "Defects4REST: A benchmark of real-world defects to enable controlled testing and debugging studies for REST APIs," in *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, 2026, pp. 1597–1608.
[49] O. Sahin, M. Zhang, and A. Arcuri, "WFC/WFD: Web fuzzing commons, dataset and guidelines to support experimentation in REST API fuzzing," *arXiv preprint arXiv:2509.01612*, 2025.
[50] P. Di, B. Liu, and Y. Gao, "MicroFuzz: An efficient fuzzing framework for microservices," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2024, pp. 216–227.
[51] C. Cao, A. Panichella, and S. Verwer, "Automated test-case generation for REST APIs using model inference search heuristic," in *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 2025, pp. 29–40.
[52] 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*, vol. 246, p. 103322, 2025.
[53] A. S. Abdelfattah, T. Cerny, J. Yero, E. Song, and D. Taibi, "Test coverage in microservice systems: An automated approach to E2E and API test coverage metrics," *Electronics*, vol. 13, no. 10, p. 1913, 2024.