MDPI

Article

Test Coverage in Microservice Systems: An Automated Approach to E2E and API Test Coverage Metrics

Amr S. Abdelfattah [1], Tomas Cerny [2]*, Jorge Yero [3], Eunjee Song [4], and Davide Taibi [5]

[1] Computer Science, Baylor University, TX, USA; amr_elsayed1@baylor.edu
[2] Systems and Industrial Engineering, University of Arizona, Arizona, USA; tcerny@arizona.edu
[3] Computer Science, Baylor University, TX, USA; jorge_yero1@baylor.edu
[4] Computer Science, Baylor University, TX, USA; eunjee_song@baylor.edu
[5] University of Oulu, Oulu, Finland; davide.taibi@oulu.fi
* Correspondence: tcerny@arizona.edu

Abstract: Test coverage is a critical aspect of the software development process, aiming for overall confidence in the product. When considering cloud-native systems, testing becomes complex since we deal with multiple distributed microservices that are developed by different teams and may change quite rapidly. In such a dynamic environment, it is important to track test coverage. This is especially relevant to end-to-end (E2E) and API testing since these might be developed by teams distinct from microservice developers. Moreover, indirection exists in E2E, where the testers see the user interface but do not know how comprehensive their test suits are. To ensure confidence in health checks in the system, mechanisms and instruments are needed to indicate the test coverage level. Unfortunately, there is a lack of such mechanisms for cloud-native systems. This manuscript introduces test coverage metrics for evaluating the extent of E2E and API test suite coverage for microservice endpoints. It elaborates on automating the calculation of these metrics with access to microservice codebases and system testing traces. It delves into the process and offers feedback with a visual perspective, emphasizing test coverage across microservices. To demonstrate the viability of the approach, we implement a proof-of-concept tool and perform a case study on a well-established system benchmark assessing existing E2E and API test suites with regard to test coverage using the proposed endpoint metrics. The results of endpoint coverage reflect the diverse perspectives of both testing approaches. API testing achieved 91.98% coverage in the benchmark, whereas E2E testing achieved 45.42%. Combining both coverage results yielded a slight increase to approximately 92.36%, attributed to a few endpoints tested exclusively through one testing approach, not covered by the other.

Keywords: microservices; end-to-end testing; API tests; test quality

Citation: Abdelfattah, A.S.; Cerny, T.; Yero, J.; Song, E.; Taibi, D. Test Coverage in Microservice Systems: An Automated Approach to E2E and API Test Coverage Metrics. *Journal Not Specified* **2024**, *1*, 0. https://doi.org/

Received:
Revised:
Accepted:
Published:

1. Introduction

Microservice architecture empowers practitioners to build scalable software systems by breaking them down into a collection of loosely coupled interacting services. Each service, responsible for a specific business capability, can be independently developed and deployed, facilitating faster development and deployment cycles, easier maintenance, and enhanced scalability.

Ensuring the robust functionality and seamless user experience of applications is crucial in software development. This necessitates the use of two distinct testing approaches: E2E testing [1], which assesses the entire application workflow by simulating real user interactions, and API testing [2], focused on verifying the reliability of the application's backend through direct interactions with its APIs [3]. Striking a balance between these approaches is essential for achieving comprehensive test coverage and delivering high-quality software products.

Conventional testing methods in this area struggle to effectively manage the complexities posed by microservices-based systems [4]. These systems have attributes such as distributed nature, continuous architectural evolution, dynamic infrastructure provisioning, and hidden complexities. This makes it challenging for conventional testing strategies since applications are broken down into smaller, interconnected microservices and deployed across various environments. Several studies highlight the absence of assessment methodologies to accommodate the microservice and distributed approaches [5,6]. This shift in paradigm requires a comprehensive reassessment of testing methodologies to ensure that microservice distributed systems meet the desired quality standards.

Testing microservice systems using either of these approaches stands with the same challenges in calculating the testing coverage for their components. In E2E testing, concealing microservice and endpoint calls within user interface interactions weakens the connection that links user interactions to the underlying endpoint calls in the system. Additionally, maintaining the sequence of testing steps proves challenging [7], especially in API testing, where deviations may occur, testing APIs designed for calls through other APIs exclusively. In essence, both approaches interact with an interface—either the user interface in E2E testing or the program interface in API testing. Both conceal the underlying logical system structure, presenting challenges in testing all possible scenarios. These challenges are compounded by testers' lack of knowledge about specific services, leading to difficulties in estimating the testing coverage for their tests [8].

Recognizing the extent to which a microservice system's individual tests involve specific microservices is crucial for testers to gain insights into system coverage and test-to-microservice dependencies. E2E tests interact with the system through the user interface, mediating interactions to the microservice endpoint level [9,10]. API tests interact through direct individual endpoint calls or composite calls that include multiple such endpoint calls. Associating tests with impacted microservices and their endpoints provides testers with insights into the comprehensiveness of their test suites in covering all system endpoints.

This paper aims to establish metrics for calculating the coverage of endpoints in E2E and API test suites, their individual tests, and microservices. It introduces a practical method and measurement approach through a case study. The automated approach proposed maps individual tests to system microservices and their endpoints, aiding testers in achieving test design completeness. By providing detailed knowledge of test-to-endpoint associations, this approach enables testers to better understand their test suite coverage and identify less apparent gaps. This paper extends our prior work [9], which aimed to establish metrics for calculating the coverage of endpoints in E2E test suites. The current paper offers a comprehensive assessment of testing approaches and includes an additional perspective on coverage calculations for API testing. Moreover, it brings a new case study illustrating differing coverages between these two types of tests, which gives more insight into how such tests stand when delivering a comprehensive perspective in terms of endpoints. Furthermore, it shows how a combined test coverage perspective could ensure better confidence in the system's health.

This paper makes the following contributions in the context of microservices:

- Expanded the proposal and evaluation of three metrics (Microservice Endpoint Coverage, Test Case Endpoint Coverage, and Complete Test Suite Endpoint Coverage) for assessing endpoint coverage in both E2E testing and API testing.
- Process of calculating metrics and implementation of proof-of-concept tool.
- A practical case study is deriving and validating the coverage metrics in a large microservice system benchmark.
- Dataset encompassing comprehensive endpoint coverage across both testing methodologies for the system benchmark.

The subsequent sections of the paper are structured as follows: Section 2 provides an elaboration on related work, while Section 3 describes the methodology, metrics, and

process. Section 4 presents a detailed case study, followed by a discussion in Section 5 and threats to validity in Section 6. The paper is concluded in Section 7.

2. Related Work

As emphasized by Horgan [11], comprehensive test coverage metrics play a crucial role in testing strategy efficacy. This notion is further supported by Whalen et al. [12], who emphasize the importance of black-box testing and the utilization of formal software requirements to thoroughly assess test suite effectiveness.

As software development progresses towards cloud-native architectures and microservices, new complexities arise in the testing landscape. Staats et al. [13] and Rajan et al. [14] explore requirements coverage metrics and their pivotal role in improving fault detection. This need for refined coverage metrics, accurately capturing the nuances of modern systems, is echoed in innovative approaches to REST API testing by Corradini et al. [15] and insights into branch coverage within continuous integration by Grano et al. [16].

Various studies have identified the lack of assessment techniques for microservice systems. A systematic literature review by Ghani et al. [5] concluded that most articles focused on testing approaches for microservices lacked sufficient assessment and experimentation. Jiang et al. [6] emphasized the need for improved test management in microservice systems to enhance their overall quality.

A recent survey by Golmohammadi et al. [17] presented the results of their systematic mapping study on testing REST APIs. They also emphasized the importance of having the right metrics to evaluate the effectiveness of the API testing and classified the state-of-the-art metrics into three types: coverage criteria, fault detention, and performance. Additionally, Waseem et al. [18] conducted a survey and revealed that unit and E2E testing are the most commonly used strategies in the industry. However, the complexity of microservice systems presents challenges for their monitoring and testing, and there is currently no dedicated solution to address these issues. Similarly, Giamattei et al. [19] identified the monitoring of internal APIs as a challenge in black box testing microservice systems, advocating for further research in this area.

To address these gaps, it is crucial to develop an assistant tool that improves system testing and provides appropriate test coverage assessment methods. Corradini et al. [20] conducted an empirical comparison of automated black-box test case generation approaches specifically for REST APIs. They proposed a test coverage framework that relies on the API interface description provided by the OpenAPI specification. Within their framework, they introduced a set of coverage metrics consisting of eight metrics (five request-related and three response-related), which assess the coverage of a test suite by calculating the ratio of tested elements to the total number of elements defined in the API. However, these metrics do not align well with the unique characteristics of microservice systems. They do not take into account the specific features of microservices, such as inter-service calls and components like API gateway testing.

Giamattei et al. [19] introduced MACROHIVE, a grey-box testing approach for microservices that automatically generates and executes test suites while analyzing the interactions among inter-service calls. Instead of using the commonly used tools such as SkyWalking or Jaeger, MACROHIVE builds its own infrastructure, which incurs additional overhead by requiring the deployment of a proxy for each microservice to monitor. It also involves implementing communication protocols for sending information packets during request-response collection. MACROHIVE employs combinatorial tests and measures the status code class and dependencies coverage of internal microservices. However, compared to our proposed approach, MACROHIVE lacks static analysis of service dependencies, relying solely on runtime data. In contrast, our approach extracts information statically from the source code, providing accurate measurements along with three levels of system coverage.

Ma et al. [21] utilized static analysis techniques and proposed the Graph-based Microservice Analysis and Testing (GMAT) approach. GMAT generates Service Dependency

Graphs (SDG) to analyze the dependencies between microservices in the system. This approach enhances the understanding of interactions among different parts of the microservice system, supporting testing and development processes. GMAT leverages Swagger documentation to extract the SDG, and it traces service invocation chains from centralized system logs to identify successful and failed invocations. The GMAT approach calculates the coverage of service tests by determining the percentage of passed calls among all the calls, and it visually highlights failing tests by marking the corresponding dependency as yellow on the SDG. However, GMAT is tailored to test microservices using the Pact tool and its APIs. In contrast, our approach introduces three coverage metrics that focus on different levels of microservice system parts, emphasizing endpoints as fundamental elements of microservice interaction. While our approach doesn't consider the status code of each test, combining GMAT with our proposed approach could offer further insights for evaluating microservice testing and assessment criteria.

Dynamic analysis supplements static analysis by utilizing instrumentation to capture and scrutinize the runtime actions of programs. This method is essential for identifying breaches of properties and understanding program behavior, as highlighted by Ball et al. [22]. The advent of NVBit, as introduced by Villa et al. [23], enhances the functionalities of dynamic binary instrumentation, enabling tailored error detection, bug identification, and performance assessment. This approach holds particular relevance in cloud-native systems, where continuous monitoring of endpoints and components in distributed architectures is critical for upholding system integrity and efficiency.

In essence, the field of software testing is experiencing a significant shift propelled by the embrace of cloud-native architectures and microservices. This shift calls for a comprehensive approach that merges E2E testing and API testing, forming the foundation for the creation of thorough coverage metrics tailored to the distinct intricacies of these systems. This paper addresses the gap in assessment techniques for microservice testing by introducing test coverage metrics and designing an analytical tool capable of evaluating microservice systems, quantifying and visualizing their test coverage.

3. Test Coverage Methodology

The objective of this methodology is to assess E2E and API testing suites in achieving coverage of endpoints within microservices-based systems. Although these testing approaches may appear distinct, this methodology outlines a generalization technique for calculating the test coverage of both. Additionally, it elucidates the specificities of how these approaches differ in their modes of interaction.

To calculate test coverage for endpoint components in a microservice system, it involves retrieving information on both the *static* endpoints declared in the system's source code and the *dynamic* endpoints actually tested during test suite execution. Subsequently, it compares these two sets of information to derive various metrics of test coverage. Thus, this methodology employs static and dynamic analysis techniques to extract the necessary information for both testing approaches.

Both static and dynamic information necessitate more specific extraction methods. Extracting static endpoints involves applying analysis techniques to the source code, which either requires customization for the programming language used, or the adoption of a polyglot technique capable of accommodating multiple programming languages in a more abstract manner, as demonstrated in [24,25]. Conversely, extracting dynamic endpoints requires analyzing traces and log information generated by specific systems such as Jaeger and Skywalking.

This methodology elaborates on each step without being tightly bound to any particular programming language or technology. This makes it sufficiently generalized to be applicable across different system setups while also being specific enough to provide clear instructions on the required attributes and techniques for calculating test coverage.

The rest of this section presents our proposed metrics and automated approach, outlining its stages for extracting the data required for calculating the metrics over systems.

3.1. Test Coverage Metrics

While testing involves test suites, each test suite contains test cases that represent a series of steps or actions defining a specific test scenario. We introduce three metrics to assess the coverage of endpoints in microservice systems: microservice endpoint coverage, test case endpoint coverage, and complete test suite coverage. These metrics are described in detail below:

- **Microservice endpoint coverage:** determines the tested endpoints within each microservice. It is obtained by dividing the number of tested endpoints from all tests by the total number of endpoints in that microservice. This metric offers insights into the comprehensiveness of coverage for individual microservices. The formula for microservice endpoint coverage is:

$$C_{\text{ms}(i)} = \frac{|E^{\text{tested}}_{\text{ms}(i)}|}{|E_{\text{ms}(i)}|} \quad ;$$

$C_{\text{ms}(i)}$ - the coverage per microservice i,
$E^{\text{tested}}_{\text{ms}(i)}$ - the set of tested endpoints in microservice i,
$E_{\text{ms}(i)}$ - the set of all endpoints in microservice i.

- **Test case endpoint coverage:** gives a percentage of endpoints covered by each test case. It is calculated by dividing the number of endpoints covered by each test by the total number of endpoints in the system. This provides insights into the effectiveness of individual tests in covering the system's endpoints. The formula for test case endpoint coverage is:

$$C_{\text{test}(i)} = \frac{|E^{\text{tested}}_{\text{test}(i)}|}{|\bigcup_j^{m\_total} E_{\text{ms}(j)}|} \quad ;$$

$C_{\text{test}(i)}$ - the coverage per test i,
$E^{\text{tested}}_{\text{test}(i)}$ - the set of tested endpoints from test i,
m_total - the total number of microservices in the system,
$\bigcup_j^{m\_total} E_{\text{ms}(j)}$ - the set of all endpoints in the system.

- **Complete Test suite endpoint coverage:** determines the test suite overall coverage of the system by dividing the total number of unique endpoints covered by all test cases in the test suite by the total number of endpoints in the system. It provides insights into the completeness of the test suite in covering all endpoints within the system. The formula for complete test suite endpoint coverage is:

$$C_{\text{suite}} = \frac{|\bigcup_i^{t\_total} E^{tested}_{\text{test}(i)}|}{|\bigcup_j^{m\_total} E_{\text{ms}(j)}|} \quad ;$$

C_{suite} - the complete test suite coverage,
m_total - the total number of microservices in the system,
t_total - the total number of tests in the test suite,
$\bigcup_i^{t\_total} E^{tested}_{\text{test}(i)}$ - the set of all tested endpoints from all tests,
$\bigcup_j^{m\_total} E_{\text{ms}(j)}$ - the set of all endpoints in the system.

3.2. Clarification Example

To provide further clarification, consider a system consisting of three microservices (MS-1, MS-2, MS-3), each with two endpoints, with a test suite composed of two tests (Test-1, Test-2), as depicted in Figure 1 for E2E test suite. In the example, the tests interact with endpoints through the user interface, which triggers the initiation of endpoint requests passed through the API gateway component. The example demonstrates that Test-1 calls two endpoints, one from MS-1 (E1.1) and one from MS-2 (E2.1). On the other hand, Test-2 calls two endpoints from MS-2 (E2.1, E2.2), and E2.2 has an inter-service call to endpoint E3.1 in MS-3. The identical illustration can be depicted in Figure 2 for the API testing suite. It showcases the same interactions; however, the calls are made directly through the API gateway component instead of the user interface.

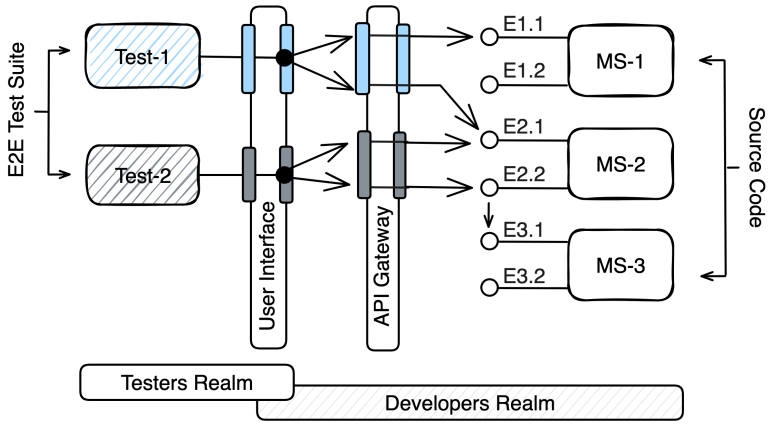


Figure 1. E2E Calculation Clarification Example

Applying our metrics on both test suites, we can calculate the microservice endpoint coverage ($C_{\text{ms}(i)}$) for each microservice. For MS-1 and MS-3, only one out of their two endpoints is tested throughout all tests, resulting in a coverage of 50% ($C_{\text{ms}(1)} = C_{\text{ms}(3)} = \frac{1}{2}$) for each. However, for MS-2, both of its endpoints are tested at least once, leading to a coverage of 100% ($C_{\text{ms}(2)} = \frac{2}{2}$).

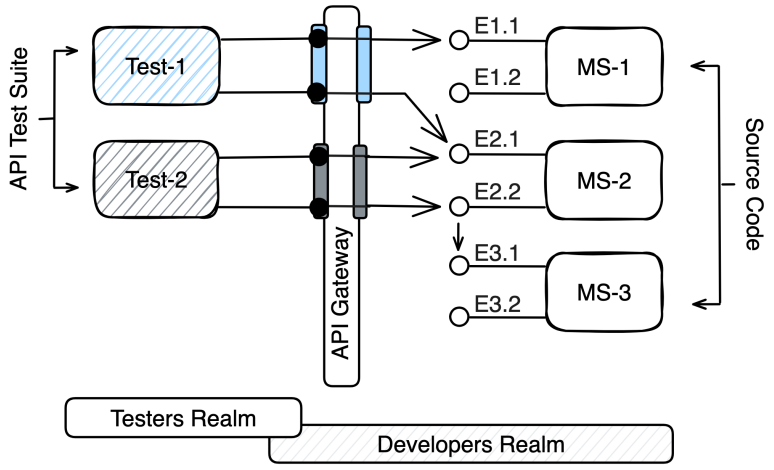


Figure 2. API Testing Calculation Clarification Example

Next, we calculate the test case endpoint coverage ($C_{\text{test}(i)}$) per each test. Test-1 covers two out of the six endpoints in the system, resulting in a coverage of approximately 33.3% ($C_{\text{test}(1)} = \frac{2}{6}$). Test-2 covers three distinct endpoints, resulting in a coverage of 50% ($C_{\text{test}(2)} = \frac{3}{6}$). It is important to highlight that Test-2 contains an inter-service call to endpoint E3.1, as shown in our approach.

Finally, we can calculate the complete test suite endpoint coverage (C_{suite}) of the system. Out of the six endpoints in the system, four distinct endpoints are tested from the two tests. This results in $\approx 66.6\%$ coverage ($C_{\text{suite}} = \frac{4}{6}$).

3.3. The Metrics Extraction Process

To automatically collect the data for calculating the test coverage metrics, we propose to employ a combination of static and dynamic analysis methods.

The static analysis phase focuses on examining the source code to extract information about the implemented endpoints in the system. The dynamic analysis phase involves inspecting system logs and traces to identify the endpoints called by the automation tests. By combining the data obtained from both analyses, the approach applies the proposed metrics to generate the endpoint coverage, and then it provides two visualization approaches to depict the coverage over the system representation. This process involves the following four stages as illustrated in Figure 3:

Stage 1. Endpoint Extraction From Source Code (Static Analysis).

Stage 2. Endpoint Extraction From Log Traces (Dynamic Analysis).

Stage 3. Coverage Calculation.

Stage 4. Coverage Visualization.

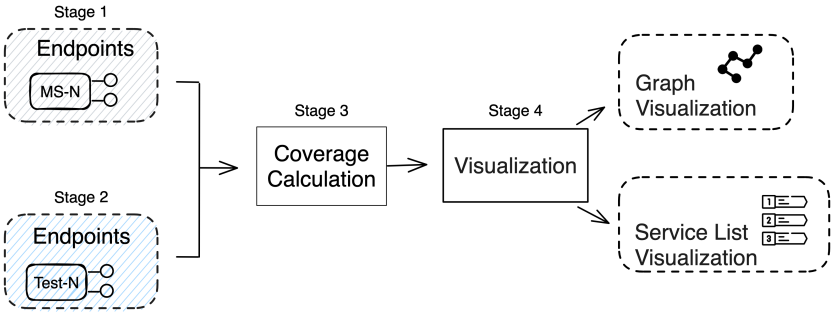


Figure 3. The proposed approach overview

We will delve into the details of each stage to demonstrate the approach.

3.3.1. Stage 1: Endpoint Extraction From Source Code (Static Analysis):

This stage aims to comprehend the offerings of the system implementation concerning the declared endpoints ready for consumption. Our approach applies a static analysis approach to the system's source code to extract the employed endpoints in each microservice ($E_{\text{ms}(i)}$). Static analysis refers to the process of analyzing the syntax and structure of code without executing it in order to extract information about the system. As depicted in Figure 4, initially, microservices can be divided and detected from the system codebase. Each microservice's codebase is then processed by the *endpoint extraction process*, which produces the endpoints corresponding to each microservice.

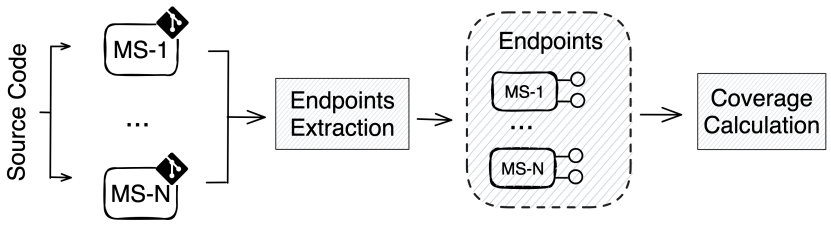


Figure 4. Stage 1: Static analysis flow

The identification of API endpoints typically relies on specific frameworks or libraries[1]. This ensures consistency in metadata identification. Code analysis extracts metadata attributes about each endpoint, including the path, HTTP method, parameters, and return type. However, identification of endpoints can be performed across platforms as demonstrated by Schiewe et al. [24] or accomplished by frameworks like Swagger[2]

As a result, a list of endpoints is generated and organized according to the respective microservice they belong to. This comprehensive list of endpoints becomes one of the inputs for our *coverage calculation process*, where it combines the output of the dynamic analysis flow.

3.3.2. Stage 2: Endpoint Extraction From Log Traces (Dynamic Analysis):

The objective of this stage is to identify the endpoints invoked by the test suites during runtime. We utilize dynamic analysis to identify the endpoints called during the execution of each test case in test suites ($E^{\text{tested}}_{\text{test}(i)}$. It also identifies the microservices containing these tested endpoints ($E^{\text{tested}}_{\text{ms}(i)}$). The analyzed system is executed to observe its runtime behavior and transactions. This analysis involves running multiple tests and capturing the traces that occur, as illustrated in Figure 5.

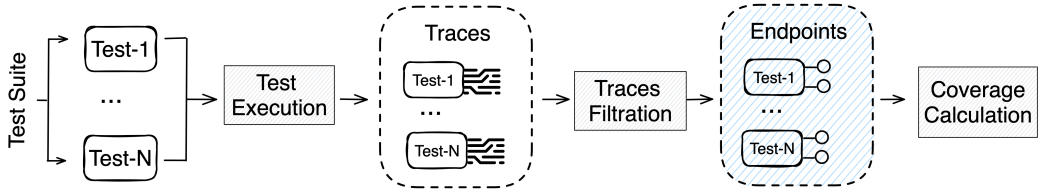


Figure 5. Stage 2: Dynamic analysis flow

The dynamic analysis flow sketched in Figure 5 has two main responsibilities. Firstly, it takes the tests (i.e., E2E tests and API tests) and executes them sequentially. During the execution of the tests, traces are generated, capturing the interactions with the system. These traces are sent to a configured centralized logging system (i.e., SkyWalking, Jaeger), which stores them in its own storage, or an externally configured data storage solution (i.e., Elasticsearch), enabling analysis and further processing. Secondly, the process calculates the delta of the produced traces to identify the traces relevant to each executed test. This can be achieved in various ways, such as recording a timestamp from the start of a test's execution to its completion, retrieving the traces after each test execution and calculating the difference based on the latest track record, or sending a dynamically generated trace before and after the execution of each test to mark the start and end. In our approach, we have employed the first strategy, as it avoids unnecessary processing and complexity at this stage.

The extracted test trace sequences corresponding to each test undergo a *traces filtration* process that filters and identifies the traces related to endpoints. This may involve queries to the trace storage to return specific trace indexes in the data. For instance, the SkyWalking tool marks the traces involving endpoint calls and makes them accessible under an index (in particular, `sw_endpoint_relation_server_side` index). Additionally, centralized logging systems encode the data records using `Base64`[3] when sending them to external storage like Elasticsearch. Therefore, this step may include an additional decoding process if needed to detect the endpoints. These endpoint-related trace records contain information about the source and destination endpoints involved in the call relationship.

---

[1] An example, in the Java Spring framework, annotations such as `@RestController` and `@RequestMapping` are commonly used.

[2] Swagger: https://swagger.io

[3] Base64: https://developer.mozilla.org/en-US/docs/Glossary/Base64

As a result, a list of endpoints is generated and organized according to the respective test suite they belong to. This list of endpoints becomes the second input for the *coverage calculation process*, where it is combined with the output of the static analysis stage.

3.3.3. Stage 3: Coverage Calculation:

During this stage, we determine which of the system endpoints have been accessed throughout the test suites and which ones have not been accessed. Therefore, this stage combines the extracted data from the previous two stages to calculate the three metrics of coverage ($C_{\text{ms}(i)}$, $C_{\text{test}(i)}$, C_{suite}). This stage follows the Set-based approach to solely account for the uniqueness of endpoints and their correspondence through the preceding stages.

> A challenge arises when matching the extracted system endpoints from the source code with those extracted from the traces. Since traces contain invoked endpoints with arguments' values (e.g., http://xxx.com/10), while those identified by static analysis hold parameter types and names (e.g., http://xxx.com/id: Integer). A similar challenge has been accounted for when profiling systems using log analysis and matching log lines with logging statements in the source code [26]. The source code contains a log message template with parameters, and execution logs contain a message with values from the execution context, which is not a direct match (i.e., source code `log.info('calling {a} from {b}')` vs. a contextual log statement `'calling for from bar'` where both `a` and `b` are interpreted). Zhao et al. [26] have identified all code log statements to extract templates that could be matched using regular expressions to identify and match the parameter types whose values are present in the log output.
>
> In our approach, we employ signature matching to solve the challenge. It involves comparing the endpoint method signature with the data and parameters exchanged during REST call communication to detect and verify the authenticity and matches of the requests. Thus, to determine which system endpoints were called by the test, we consider the comparison of extracted attributes of the endpoints (such as path, request type, and parameter list) from the source code with the REST calls extracted from the test traces. This matching process helps to establish the coverage levels and determine which endpoints were invoked by the tests.

The calculation of $C_{\text{ms}(i)}$ involves categorizing and dividing the number of tested endpoints (Stage 2) by the number of declared endpoints for each microservice (Stage 1). For $C_{\text{test}(i)}$, the calculation entails extracting, for each test case, the number of endpoints covered (Stage 2), and dividing it by the total number of endpoints in the system (Stage 1). This computation reveals the percentage of coverage that a test case achieves across the entire system's endpoints. Finally, C_{suite} is determined by dividing the total number of distinct endpoints covered by all test cases in the test suite (Stage 2) by the total number of endpoints in the system (Stage 1).

3.3.4. Stage 4: Coverage Visualization:

While microservice architecture primarily caters to large systems, it is essential to provide a user-friendly presentation to assist practitioners in easily comprehending the coverage within the familiar context of the system. Therefore, this approach provides a centralized visualization of the coverage calculation by offering two methods for visualizing these coverage metrics. The first displays a list of microservices, with each microservice showing its endpoints. Covered endpoints are marked in green, while missed endpoints are marked in red, as demonstrated in Figure 12a. The second representation utilizes the service dependency graph, where microservices are represented as nodes, and the dependencies between them are shown as edges. The nodes in the graph are color-coded based on the coverage percentage, allowing users to visually observe the coverage on the

holistic system view depicting service dependencies, as exampled in Figure 12b. These visualization techniques help in interpreting the two metrics of $C_{\text{ms}(i)}$ and $C_{\text{test}(i)}$. Thus, these coverage calculations and visualizations provide valuable insights into the extent of test coverage achieved by automation frameworks in the context of microservices, enabling users to visually assess the effectiveness of their testing efforts and identify areas that require improvement.

3.4. Methodology Discussion

This methodology elucidates the interplay between static data analysis and dynamic data analysis, which are pivotal for calculating testing coverage metrics. Its design ensures versatility for polyglot systems, offering detailed guidelines without strict adherence to any specific programming language or technology. This flexibility facilitates broad applicability across diverse system configurations while maintaining specificity in guiding the calculation of test coverage. Consequently, implementation details may vary between system environments to accommodate the methodology effectively.

However, challenges arise when reconciling information extracted from static and dynamic phases, as they exhibit distinct characteristics. Mismatched endpoint signatures between source code and traces can occur due to discrepancies in trace values aligning with defined types in the code. Consequently, the methodology acknowledges and addresses this challenge to ensure accuracy in matching extracted data.

Despite differences in communication layers between E2E testing and API testing —where E2E testing traverses various layers from the user interface to API-gateway to endpoints, while API testing focuses on specific endpoint calls that may pass through API-gateway directly— the methodology remains applicable for calculating testing coverage in both scenarios. By extracting traces generated during execution, regardless of the communication layers traversed, the methodology captures comprehensive endpoint testing coverage. Moreover, its flexibility enables focused analysis of testing specific communication layers, thereby highlighting inter-service communication calls and distinguishing tests passing through the API-gateway from those bypassing it. This distinction underscores the importance of the API-gateway as a filtration point, particularly for enforcing cross-cutting aspects such as security authentication.

Furthermore, implementing this methodology across both testing approaches offers substantial assurance of system health from diverse perspectives. Each approach targets distinct testing strategies to ensure system testability, such that E2E testing focuses on user scenario perspective, while API testing emphasizes functionality reliability.

4. Case Study

In this section, a case study is conducted to showcase the feasibility of the proposed automated metric calculation approach. The objective is to provide testers with insights that enhance system coverage and testability. This is achieved by implementing the proposed approach and its stages to extract the necessary data for calculating the three coverage metrics. The case study involves integrating data extraction from the system source code with log traces generated during the execution of both E2E and API test suites.

This case study considered an open-source system benchmark and utilized an existing E2E test suite and API test suite designed for the same system. A proof of concept (POC) was developed to illustrate the automation of the proposed metrics calculation, which was employed to assess the provided test benchmarks. The complete data analysis phases with their results are published in a dataset[4]. This dataset contains the complete calculations of the metrics.

[4] Dataset: https://zenodo.org/records/10553186

4.1. Proof of Concept (POC) Implementation

This section describes the implementation of a POC[5] to showcase the four phases of the proposed approach. We focused on statically analyzing Java-based project source codes that use the Java Spring Cloud framework, an open-source framework that is widely used for building cloud-native applications. It provides developers with a comprehensive set of tools and libraries to build scalable and resilient applications in the Java ecosystem.

For the endpoint extraction from source code (Stage 1), we utilized the open-source JavaParser library [27]. It allowed us to parse Java source code files, generate an Abstract Syntax Tree (AST) representation, and traverse it to detect spring annotations such as `@GetMapping` and `@PostMapping`. We extracted the relevant attributes once the endpoints were detected.

For the endpoint extraction from log traces (Stage 2), we focused on extracting the skywalking generated logs and traces from Elasticsearch, which is widely adopted as a central component in the ELK (Elasticsearch, Logstash, Kibana) stack [28]. We used the Elasticsearch Java High-Level REST Client [29] which offers a convenient way to interact with Elasticsearch. It provided a QueryBuilder class to construct queries for searching and filtering data, such as creating a query to retrieve the logs that are between specific start and end timestamps.

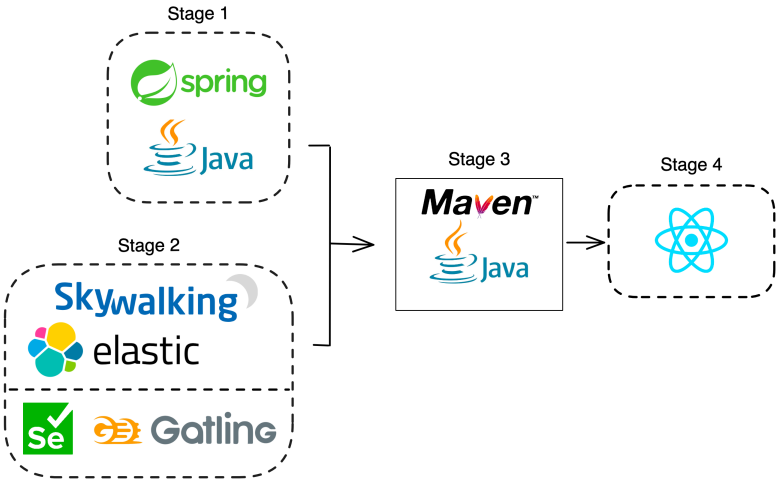


Figure 6. Frameworks Considered in the POC

Then, the POC performs the coverage calculation (Stage 3). It utilized Apache Maven, a build automation tool for Java projects, to execute the test suites of Selenium [30] and Gatling [31]. After that, it integrates the results of the two stages' outcomes of static and dynamic processes and applies the proposed metrics. For the coverage visualization (Stage 4), we provided the two visualization approaches discussed earlier. We employed React framework to implement a web application[6] that presents the information in an expandable list view for easy navigation. To integrate with the service dependency graph visualization, we utilized the Prophet library[7], an open-source project that generates the graph from source code. Additionally, we utilized the visualizer library[8], which offers a tailored 3D microservices visualization for service dependency graphs. The summarized frameworks for each stage are depicted in Figure 6.

[5] POC Source Code: https://github.com/cloudhubs/test-coverage-backend
[6] Coverage Visualizer: https://github.com/cloudhubs/test-coverage-frontend
[7] Prophet: https://github.com/cloudhubs/graal-prophet-utils
[8] 3D Visualizer: https://github.com/cloudhubs/graal_mvp

4.2. Benchmark and Test Suites

To ensure unbiased testing of our application, we utilized an open-source testbench consisting of the TrainTicket system and associated test suites.

TrainTicket [32,33] is a microservice-based train ticket booking system that is built using the Java Spring framework. It uses the standard annotations for defining the endpoints and uses the *RestTemplate* Java client to initiate requests to endpoints. This benchmark consists of 41 Java-based microservices and makes use of Apache SkyWalking [34] as its application performance monitoring system.

In order to run the TrainTicket system and execute tests on it, certain configuration fixes were necessary. To address this, a fork[9] of the TrainTicket repository was created, specifically from the 1.0.0 release. This fork incorporated the necessary fixes and a deployment script. TrainTicket integrates with Elasticsearch, allowing our POC to utilize SkyWalking to forward system logs to Elasticsearch for additional processing and analysis.

For the test suites, we utilized an open-source test benchmark[10] published in [35]. This benchmark aims to test the same version of the TrainTicket system. It contains 11 different E2E test cases using the Selenium framework and 26 API test cases using the Gatling framework.

4.3. Ground Truth

To validate the completeness of our approach, we performed a manual analysis to construct the ground truth for the test benches. The complete results of the ground truth are published in the open accessed dataset[4]. This involved manual extraction of the data related to the first two stages in our proposed process in Section 3.3, as follows: endpoint extraction from source code and endpoint extraction from log traces.

For Stage 1, we validated the endpoints extracted during the static analysis by manually inspecting the source code of the microservices' controller classes. This allowed us to identify and extract information such as the endpoint's path, request type, parameter list, and return type. This process extracted 262 defined endpoints in the TrainTicket testbench codebase.

For Stage 2, we verified the endpoints identified during the dynamic analysis by reviewing both the E2E (Selenium) and API (Gatling) test suites. Given that E2E tests primarily involve UI-based interactions and do not explicitly mention endpoints, we conducted a manual analysis of the logs generated by these tests stored in Elasticsearch. The logs contained encoded details about source and destination endpoints, which we decoded and filtered to compile a list of 171 unique endpoints invoked during E2E tests and 495 unique endpoints included during API tests. These unique endpoints encompass non-actual system endpoints, such as API gateway mediator calls, which will be filtered out in the next steps.

4.4. Case Study Results

We began the execution by running the deployment script to set up the TrainTicket system on a local instance. Subsequently, our POC executed the test cases from the provided test benchmarks, generated the list of called endpoints, and calculated the test coverage according to the described metrics for each of the E2E and API tests separately. The execution of the POC takes a few seconds to extract the data and calculate the metrics.

In terms of evaluating the completeness of our POC, this case study confirmed that we captured all the endpoints declared in the ground truth. The POC successfully captured all 262 implemented endpoints in the system, demonstrating the completeness of the Stage 1 outcome. For Stage 2 completeness, the POC extracted all 171 endpoints during E2E tests and 495 unique endpoints during API testing. The execution of the POC and the detailed

[9] TrainTicket: https://github.com/cloudhubs/train-ticket/tree/v1.0.1
[10] Test benchmark: https://github.com/cloudhubs/microservice-tests

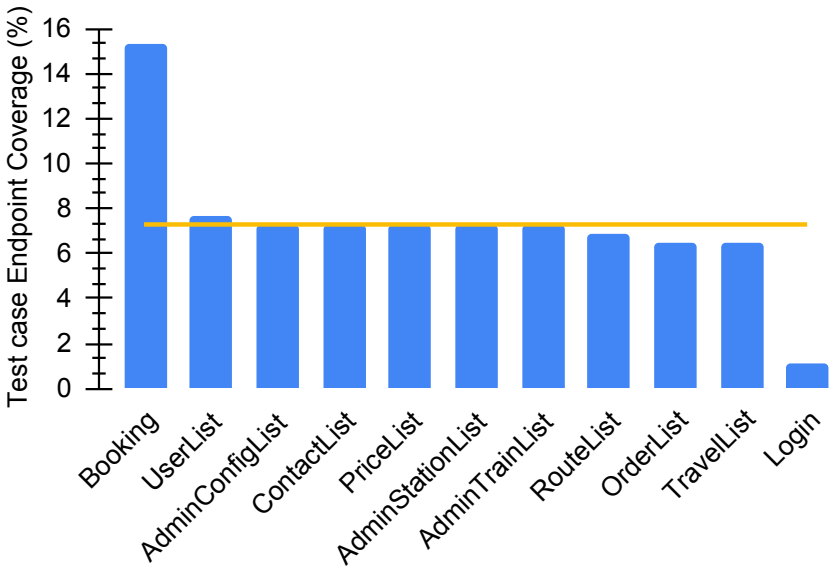


Figure 7. Test case Endpoint Coverage in the E2E testing Benchmark ($C_{\text{test}(i)}$)

coverage metrics calculations (Stage 3) are outlined in the following subsections, including both E2E testing using Selenium and API testing using Gatling.

4.4.1. End-to-End Testing Results (Selenium)

The results of the experiment execution on the E2E tests benchmark revealed a total of 171 unique endpoints extracted from a set of 953 log records[4] generated during the execution of the test cases, out of which 119 endpoints are actual endpoints within the system, and 52 endpoints are related to API gateway calls.

Through the complete data extraction, we calculate the complete test suite coverage to be approximately 45.42% ($C_{\text{suite}} = \frac{119}{262} \approx 45.42\%$). The summary statistics for the metrics calculations are provided in Table 1.

The calculation of $C_{\text{test}(i)}$ shows that the maximum coverage achieved by a test case in the study is approximately 15.27%. This was observed in the `Booking` test case, which made 53 calls to 40 unique endpoints in the system. On the other hand, the minimum coverage is approximately 1.14%, which occurred in the `Login` test case that only called three endpoints. The analysis shows that the average test case endpoint coverage is approximately 7.29%, while the most common coverage among the test cases is approximately 7.25%. This coverage was observed in the following five test cases: `AdminConfigList`, `ContactList`, `PriceList`, `AdminStationList`, and `AdminTrainList`. Figure 7 illustrates the endpoint coverage achieved by the 11 test cases, along with the average coverage for better measurement.

The calculation of $C_{\text{ms}(i)}$ reveals that the maximum coverage is 100%, observed in the `ts-verification-code-service`, which has two endpoints covered by the test cases. On the other hand, the minimum coverage is 0%, indicating that test suites completely missed testing any endpoints in the following four microservices: *ts-wait-order-service*, *ts-preserve-*

Table 1. End-to-End Summary Statistics of Coverage Metrics

Metric	**Coverage (%)**			
$C_{\textbf{suite}}$	45.42			
	Minimum	**Average**	**Maximum**	**Mode**
$C_{\textbf{ms}(i)}$	0	44.5	100	25
$C_{\textbf{test}(i)}$	1.14	7.29	15.27	7.25

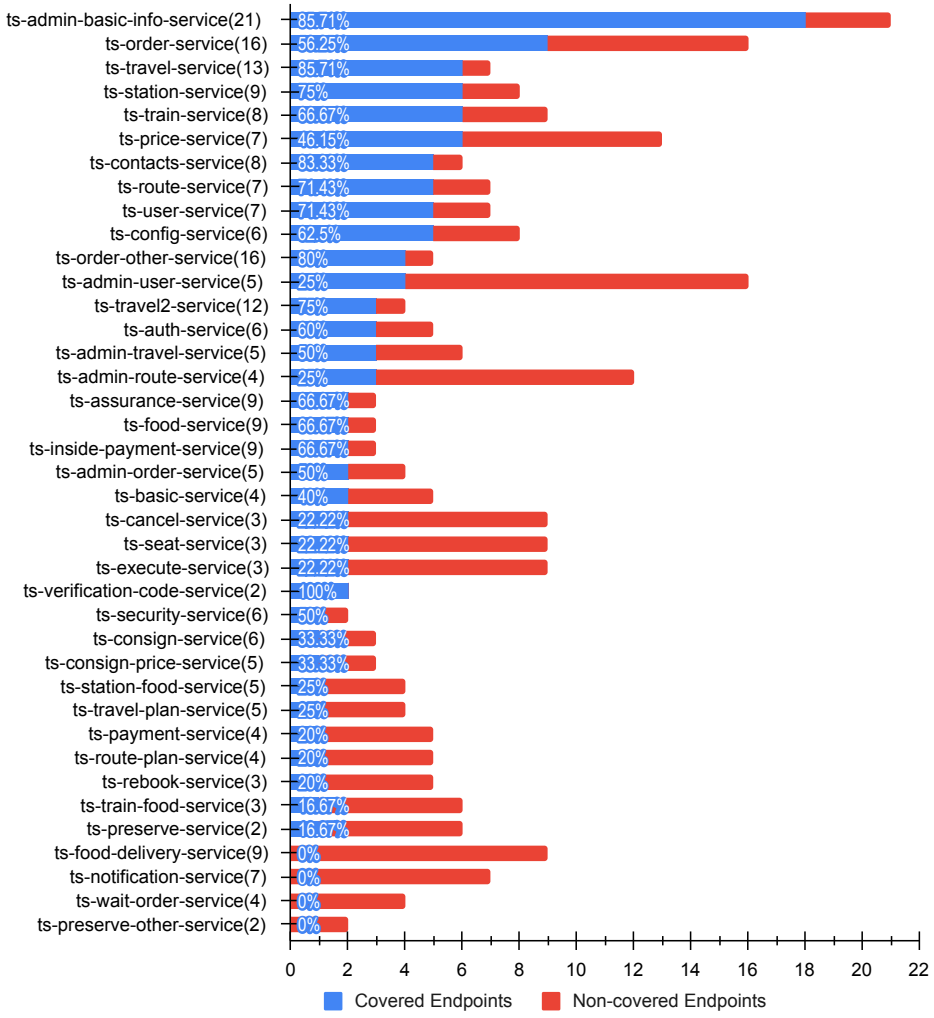


Figure 8. Microservice Endpoint Coverage in the E2E testing Benchmark ($C_{ms(i)}$)
The numbers in parentheses indicate the total number of endpoints in each ms.

other-service, *ts-notification-service*, and *ts-food-delivery-service*. The average microservice endpoint coverage is approximately 44.5%, while the mode statistics show that 25% is the most common coverage, observed in the following four microservices: *ts-travel2-service*, *ts-payment-service*, *ts-route-plan-service*, and *ts-order-other-service*. The complete calculations for each microservice are illustrated in Figure 8.

4.4.2. API Testing Results (Gatling)

The outcomes of the experiment execution in the API tests benchmark unveiled a total of 495 distinct endpoints extracted from a collection of 1902 log records[4] generated during the test case execution. Among these, 241 endpoints correspond to actual endpoints within the system, 249 endpoints are associated with API-gateway calls that are not actual endpoints in the system, and 5 endpoints deviate from the correct API signature in declared the system, as detailed in Table 2.

Additionally, we manually retrieved endpoints from the API tests implementation since they are explicitly referenced in the source code of the tests in contrast to the case in E2E tests. This enables us to perform additional validation with API testing for endpoints do not appear in the logs but may be integrated into the tests source code. This revealed that two microservices (*ts-wait-order-service* and *ts-food-delivery-service*) are slated for testing, but they do not surface in the logs due to misconfigurations in the TrainTicket testbench system. Consequently, this highlights an additional aspect that our methodology can reveal

Table 2. API testing's Endpoints Mismatched with System Signatures

Mis-matched Endpoint	Reason
ts-consign-price-service/api/v1/consignpriceservice/ consignprice@GET	Expected POST method
ts-auth-service/api/v1/users/login@GET	Expected POST method
ts-security-service/api/v1/securityservice/ securityConfigs@DELETE	Expected 1 parameter
ts-assurance-service/api/v1/assuranceservice/types@GET	Expected '/assurances'
ts-assurance-service/api/v1/assuranceservice/assurances/ orderid/{id}@GET	Mis-spelled '/assurance'

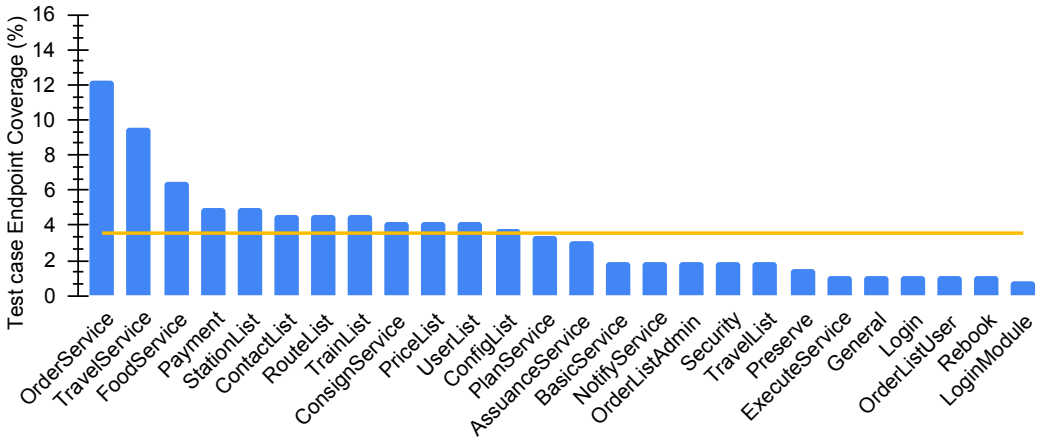


Figure 9. Test case Endpoint Coverage in the API testing Benchmark ($C_{\text{test}(i)}$)

by comparing the expected coverage derived from the endpoints in tests with the actual coverage derived from the logs. Through the complete data extraction, we calculate the complete test suite coverage to be approximately 91.98% ($C_{\text{suite}} = \frac{241}{262} \approx 91.98\%$). The summary statistics for the metrics calculations are provided in Table 3.

The computation of $C_{\text{test}(i)}$ indicates that the highest coverage attained by a test case in the study is around 12.21%. This was evident in the `OrderService` test case, which covered 32 unique endpoints in the system. Conversely, the minimum coverage is approximately 0.76%, observed in the `LoginModule` test case that only invoked two endpoints. The analysis reveals that the average test case endpoint coverage is about 3.55%, with the most common coverage among the test cases being around 1.90%. This coverage was observed in the following five test cases: `BasicService`, `NotifyService`, `OrderListAdmin`, `Security`, and `TravelList`. Figure 9 illustrates the endpoint coverage achieved by the 26 test cases, along with the average coverage for better measurement.

The calculation of $C_{\text{ms}(i)}$ discloses that the maximum coverage is 100%, which is also the most prevalent coverage for 32 microservices in the system. Conversely, the minimum coverage is 0%, signifying that the test suites entirely overlooked testing any endpoints

Table 3. API Testing Summary Statistics of Coverage Metrics

Metric	**Coverage (%)**			
$C_{\textbf{suite}}$	91.98			
	Minimum	**Average**	**Maximum**	**Mode**
$C_{\textbf{ms}(i)}$	0	91.77	100	100
$C_{\textbf{test}(i)}$	0.76	3.55	12.21	1.90

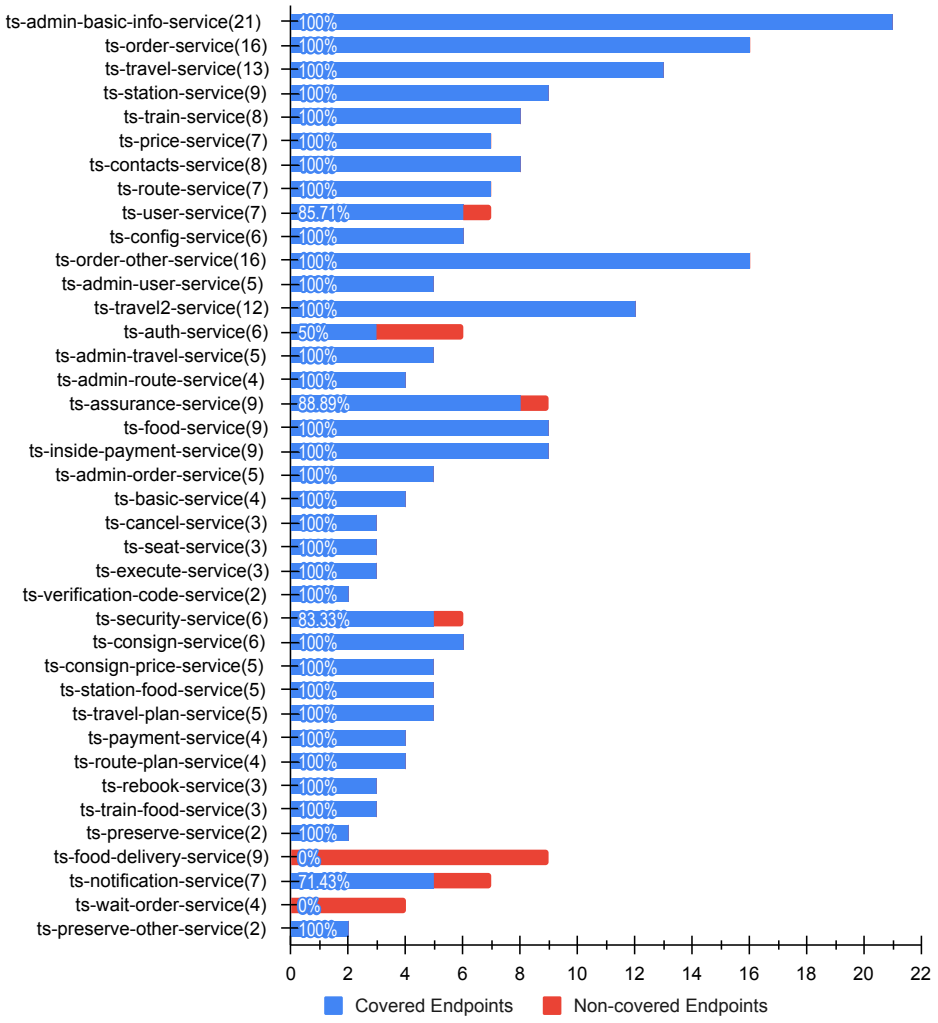


Figure 10. Microservice Endpoint Coverage in the API testing Benchmark System ($C_{ms(i)}$)
The numbers in parentheses indicate the total number of endpoints in each ms.

in the following two microservices: *ts-wait-order-service* and *ts-food-delivery-service*. The average microservice endpoint coverage is approximately 91.77%. The detailed calculations for each microservice are depicted in Figure 10.

4.5. On Combined E2E and API Test Coverage

While E2E tests ensure that the user-facing aspects of the system work as intended, the API tests validate the functionality and communication between backend services. Combining the coverage generated from both test suites is expected to yield a more comprehensive overview of the assurance or confidence in the system's health at the granularity of system endpoints. Such combined tests ensure that endpoints are responsive and touched by at least some tests.

When a problem is detected, it is still relevant to perform both E2E and API tests as they help isolate the issue. For instance, they can determine whether it's a frontend, backend, or integration problem, making debugging and fixing more efficient. Combining E2E and API tests creates a robust testing strategy that addresses different aspects of the system, leading to improved reliability and faster identification of issues. Thus, for such a perspective, a combined test coverage becomes relevant to the comprehensive evaluation of the system.

In terms of microservice endpoint coverage ($C_{ms(i)}$), the combined approach only increased the coverage of the *ts-assurance-service* microservice, reaching 100% coverage

compared to the 88.88% achieved with API testing coverage and 22.22% achieved with E2E testing coverage. This improvement is attributed to E2E tests successfully covering a misconfigured endpoint of *ts-assurance-service/api/v1/assuranceservice/assurances/types@GET* in the API tests. However, the coverage for the remaining 40 microservices in the system remains unchanged from API test coverage.

Conversely, the combination of $C_{\text{test}(i)}$ in both E2E and API test coverage involves a simple appending process since each test suite serves a different purpose with distinct testing objectives from the API test and E2E test. Thus, this shows a more comprehensive list of tests and their attached coverage of the system.

The complete test suite endpoint coverage (C_{suite}) experienced a slight increase to approximately 92.36% ($C_{\text{suite}} = \frac{242}{262} \approx 92.36\%$) after combining the endpoints from E2E and API tests. This contrasts with the individual coverage percentages of 91.98% for API and 45.42% for E2E. The intersection between endpoints covered from each of these two testing approaches was calculated, as depicted in Figure 11. It reveals that both the E2E and API test suites collectively covered a total of 118 endpoints. Moreover, the E2E test suite specifically covered an endpoint (*ts-assurance-service/api/v1/assuranceservice/assurances/types@GET*) that was not addressed by the API test suites. Conversely, the API test suites covered an additional 123 endpoints that were not included in the E2E test suite's coverage. Consequently, in total, the TrainTicket system had 21 endpoints that were not addressed by either of the test suites.

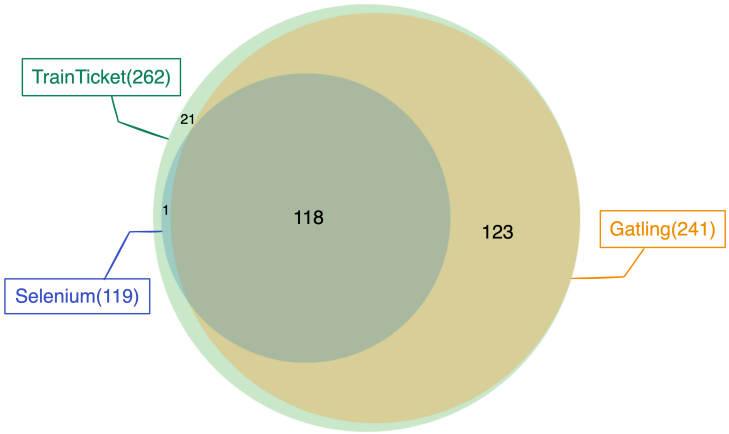
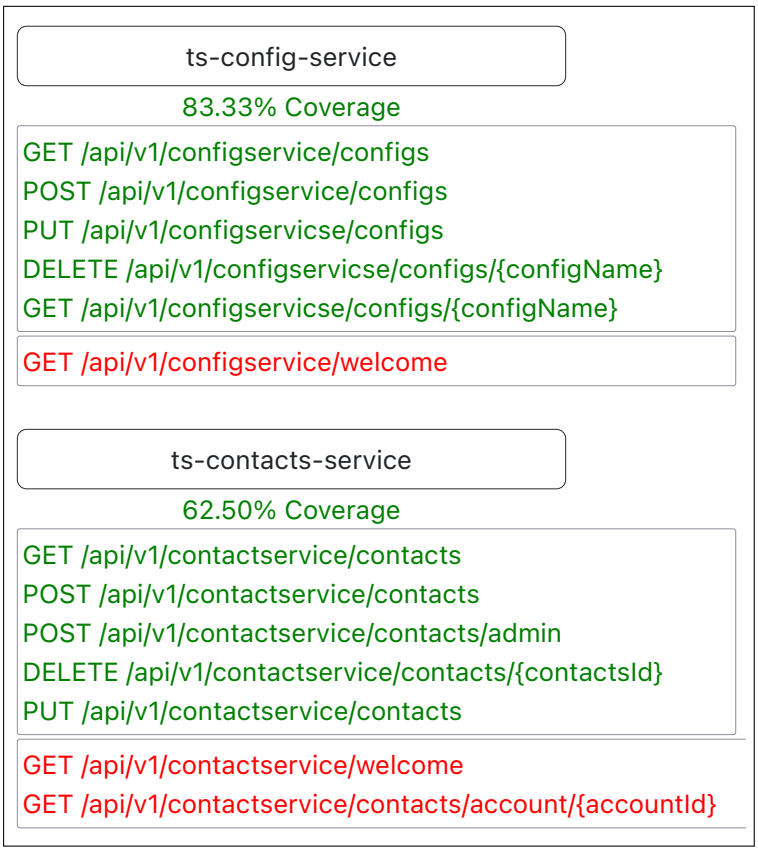


Figure 11. Combined Endpoint Coverage in the E2E and API tests Benchmark

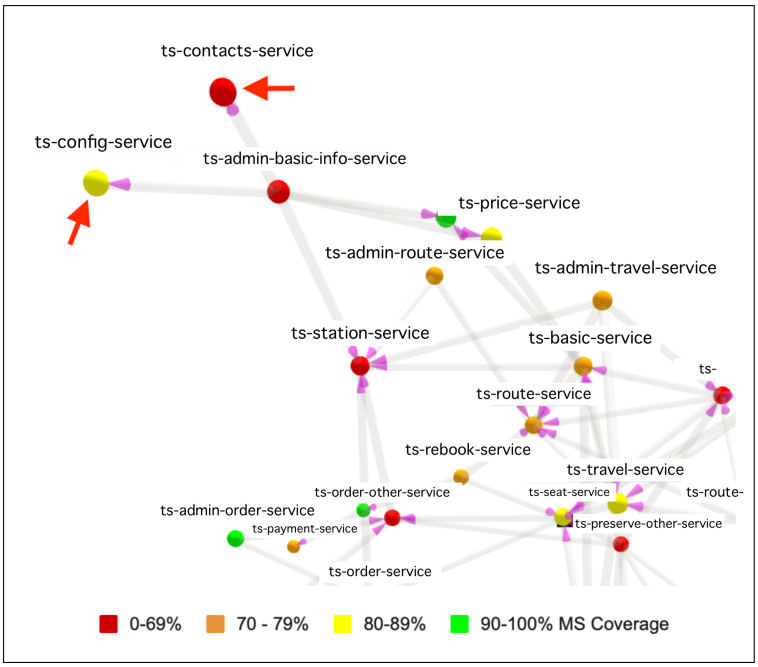
4.6. Coverage Visualization

The metrics calculations are visualized using two visualization approaches (Stage 4), as shown in Figure 12a and Figure 12b. One has a service list view, and the other provides a holistic service dependency overview in the context of endpoint coverage. The service list view consists of multiple expandable lists to present a comprehensive display of all microservices within the system, as depicted in Figure 12a. Each expandable list header includes the microservice name and its coverage percentage, while the body exhibits the paths of endpoints associated with that microservice. This visualization employs red-green color coding, such that covered endpoints are highlighted in green, and uncovered ones are marked in red. For instance, the `ts-config-service` microservice shows an approximate coverage of 83.33% from the E2E test suite, missing only one (*GET@/api/v1/configservice/welcome*) out of six endpoints. In contrast, the `ts-contacts-service` has an approximate coverage of 62.50%, with two out of seven endpoints remaining untested.

On the other hand, the service dependency view utilizes a 3D graph visualization representing the complete service dependency graph of the system, as illustrated in Figure 12b. Nodes represent microservices, and edges denote dependencies between microservices. This approach introduces four color codes based on microservice coverage percentages: red



(a) Expandable list view shows microservice endpoints list



(b) 3D interactive visualizer shows service dependencies (cropped view)

Figure 12. Microservices endpoint coverage visualization (full pictures[6])

for 0-69%, orange for 70-79%, yellow for 80-89%, and green for 90-100%. For example, the node corresponding to `ts-config-service` is highlighted in yellow, indicating its coverage of 83.33%. In contrast, the node for `ts-contacts-service` is marked in red, representing

its coverage of 62.50%. This 3D graph visualization provides a dynamic representation of service dependencies along with their respective coverage statuses.

5. Discussion

Our approach brings a promising solution to maintaining system reliability through better assurance of E2E and API test suit completeness. It contributes to the continuous reliability and quality assurance of decentralized microservice systems. In addition to integrating both testing methodologies, our approach, in contrast to existing literature, takes into consideration specific features of microservice architecture, including inter-service communication and components such as API gateway testing. Many existing studies overlook these microservice-specific characteristics. Furthermore, our approach offers three levels of granularity, enabling developers and testers to identify and benefit from the specific parts requiring modification as the system evolves. Our assessment results indicate a positive impact on establishing connections between different tests and system endpoints through automated means. Such tracking provides valuable insights for testers in managing change propagation to the testing infrastructure, as they directly indicate co-change dependencies between specific microservices or endpoints and particular tests.

Combining E2E testing with API testing provides an even more comprehensive perspective on system coverage. However, it is essential to consider the context in which the approach is applied, recognizing that the user interface in E2E testing may not interact with all middleware endpoints. This can be reflected in the provided metrics, indicating that the E2E test might not achieve 100% coverage. This prompts the question of whether the remaining endpoints signify the presence of the *Nobody Home* smell [36], indicating missing wiring from the user interface, or if they represent outdated or dead code.

On the contrary, API tests in our study covered a substantial portion of these endpoints, resulting in higher coverage compared to E2E tests. Nonetheless, API testing could include tests of deprecated, removed, or unused endpoints from the user interface perspective, as it directly points to API endpoints for testing. Nevertheless, the advantage of API testing is rooted in its static source code, which encompasses the tested endpoint APIs. This enables the approach to identify directly declared endpoints within the API test suite source code. The approach can cross-reference the declared endpoints in API test suites with those extracted from the system source code, pinpointing those that no longer exist in the system. Moreover, analyzing the log traces to distinguish covered endpoints from direct user calls (via the user interface or direct API calls) versus those covered through inter-service calls (from another microservice) adds an extra layer of validation. This aids practitioners in ensuring the design and exposure perspective of each endpoint in the system.

Furthermore, considering the nature of the testing approaches in the case study, the API tests exhibited higher coverage in microservices and complete test suite metrics, while the metric of test case endpoint coverage showed lower coverage per test case. This discrepancy arises because API tests executed a larger number of test cases, each consuming fewer endpoints, while E2E test cases consumed more endpoints each but constituted a smaller number of test cases in the overall test suite.

It is worth noting that microservices often implement `isAlive/welcome` endpoints for health checks. Some libraries, like Hystrix, can automatically generate these endpoints, while others may implement them manually. In the case of TrainTicket, 39 endpoints were implemented that were not utilized in the user interface since rendering them meaningless. However, they are considered for testing in the API tests. Verifying these endpoints can ensure that the system is correctly initialized.

6. Threats to Validity

In this section, we address the potential validity threats to our approach. We adopt Wohlin's taxonomy [37], which encompasses construction, external, internal, and conclusion threats to validity, as a framework for our analysis.

A potential **construction validity threat** arises from the dependency on static analysis for endpoint extraction and dynamic analysis of centralized traces generated by tests. It includes missing or non-standard source code and a lack of support for centralized traces, which can hinder our approach.

Our POC is currently implemented for specific programming languages and frameworks. However, it is important to note that the methodology itself is not limited to these specifications. It can be adapted and applied to other languages and frameworks, mitigating construction threats related to dependencies. Moreover, asynchronous messaging poses a potential risk to test execution by causing ghost endpoint call trace events. To mitigate this threat, potential approaches include disabling asynchronous services or conducting repeated test executions to minimize the impact.

Internal validity threats arise from potential mismatches between the extracted endpoint signatures from the source code and the traces. Although overloads are infrequent, inaccurate matching may occur due to trace values not aligning precisely with the defined types in the code. For example, if a trace contains an integer in the URL, it may match with an integer parameter type even if the corresponding endpoint has a string parameter type. Moreover, Multiple authors collaborated to ensure accurate data and calculations. They independently verified and cross-validated the results, rotating across validation processes to minimize learning effects.

To address **external validity threats**, our case study utilized a widely recognized open-source benchmark to evaluate its endpoints coverage using our proposed approach. Still, it is important to acknowledge that the results and conclusions drawn from this specific benchmark may not fully represent the entire range of microservices systems that adhere to different standards and practices.

One potential **conclusion validity threat** is that our tool was tested on an open-source project rather than an industry project. However, we aimed to address this by selecting an open-source project that employed widely-used frameworks in the industry. Furthermore, to ensure the reliability and consistency of our results, we performed the case study in multiple environments and confirmed that the outcomes remained consistent.

7. Conclusion

Test coverage is an important part of software development. The lack of tools to provide feedback on test coverage leaves an open gap for cloud-native and microservice-based systems. This work proposes endpoint-based metrics for E2E and API test coverage of such systems. Moreover, it illustrates an automation approach to extract such metrics and evaluates the approach through a proof-of-concept implementation assessed on a case study using a third-party system. Such a mechanism can provide testers with an important perspective of how complete the test coverage is with respect to the number of endpoints in the system that are involved in tests.

Furthermore, the presented approach establishes connections between tests and microservice endpoints at three distinct levels. It showcases the coverage of the entire suite of tests on each individual microservice, the coverage of each test case across the entire system endpoints, and the coverage of the entire suite of tests on the complete set of system endpoints. Additionally, It demonstrated two approaches for visualizing microservice test coverage within the holistic context.

The results of the case study highlighted distinct outcomes from both the E2E and API test suites applied to the same microservice benchmark. The API tests exhibited a high coverage percentage, which is reasonable given their focus on targeting specific APIs in their testing. However, they lacked a realistic sequence of calls that mirror real-world scenarios from the user's perspective. Conversely, the E2E test suites established this realistic chain of calls starting from the user interface through to the system endpoints. Nevertheless, they demonstrated a lower test coverage percentage of the system. This could be attributed to unused endpoints within the system user interface or insufficient tests in the suites to cover the entire user interface scenario. Combining the coverage generated by

both test suites is anticipated to provide a more comprehensive assessment of the system's assurance of endpoints. This is particularly relevant considering that each test approach aims to achieve distinct objectives, and their integration can offer a more holistic view of the system's health and functionality coverage.

While there are missing tools in decentralized systems that could provide feedback on test coverage for tests that involve the system as a whole if we resort to system endpoint coverage, a feasible mechanism can be provided by automated means and still provide relevant feedback. Yet, if endpoints provide a broad range of conditional executions, this approach will have limited descriptive value since it only measures if an endpoint was reached in test execution.

In future work, we aspire to explore the evolution of both the system and the test suite, delving deeper into the details beneath endpoints. Furthermore, we intend to expand our metrics to include a wider range of test paths within the endpoints. Additionally, we envision conducting more comparative studies and integrating with existing literature to provide more comprehensive instruments for the community.

Author Contributions: Conceptualization, Amr S. Abdelfattah, Jorge Yero; methodology, Amr S. Abdelfattah, Tomas Cerny, validation, Tomas Cerny, Eunjee Song, Davide Taibi; formal analysis, Amr S. Abdelfattah; investigation, Amr S. Abdelfattah, Tomas Cerny, Jorge Yero; resources, Amr S. Abdelfattah; data curation, Amr S. Abdelfattah, Jorge Yero; writing–original draft preparation, Amr S. Abdelfattah, Tomas Cerny; writing–review and editing, Amr S. Abdelfattah, Tomas Cerny, Jorge Yero, Eunjee Song, Davide Taibi; visualization, Amr S. Abdelfattah supervision, Tomas Cerny; project administration, Tomas Cerny; funding acquisition, Tomas Cerny, Davide Taibi.

Funding: This material is supported by the National Science Foundation under Grant No. 2409933 and Grant No. 349488 (MuFAno) from the Academy of Finland.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Tsai, W.T.; Bai, X.; Paul, R.; Shao, W.; Agarwal, V. End-to-end integration testing design. In Proceedings of the 25th Annual International Computer Software and Applications Conference. COMPSAC 2001. IEEE, 2001, pp. 166–171.
2. Ehsan, A.; Abuhaliqa, M.A.M.; Catal, C.; Mishra, D. RESTful API testing methodologies: Rationale, challenges, and solution directions. *Applied Sciences* **2022**, *12*, 4369.
3. Sharma, A.; Revathi, M.; et al. Automated API testing. In Proceedings of the 2018 3rd International Conference on Inventive Computation Technologies (ICICT). IEEE, 2018, pp. 788–791.
4. Bhojwani, R. Design patterns for microserviceto-microservice communication-dzone microservices, 2018.
5. Ghani, I.; Wan-Kadir, W.M.; Mustafa, A.; Imran Babir, M. Microservice Testing Approaches: A Systematic Literature Review. *International Journal of Integrated Engineering* **2019**, *11*, 65–80.
6. Jiang, P.; Shen, Y.; Dai, Y. Efficient software test management system based on microservice architecture. In Proceedings of the 2022 IEEE 10th Joint International Information Technology and Artificial Intelligence Conference, 2022, Vol. 10, pp. 2339–2343.
7. Jorgensen, A.; Whittaker, J.A. An api testing method. In Proceedings of the Proceedings of the International Conference on Software Testing Analysis & Review (STAREAST 2000), 2000.
8. Raj, P.; Vanga, S.; Chaudhary, A. *Cloud-native computing: How to design, develop, and secure microservices and event-driven applications*; John Wiley & Sons, 2022.
9. Abdelfattah, A.S.; Cerny, T.; Salazar, J.Y.; Lehman, A.; Hunter, J.; Bickham, A.; Taibi, D. End-to-End Test Coverage Metrics in Microservice Systems: An Automated Approach. In Proceedings of the Service-Oriented and Cloud Computing; Papadopoulos, G.A.; Rademacher, F.; Soldani, J., Eds., Cham, 2023; pp. 35–51.
10. Abdelfattah, A.S.; Cerny, T. Roadmap to Reasoning in Microservice Systems: A Rapid Review. *Applied Sciences* **2023**, *13*. https://doi.org/10.3390/app13031838.
11. Horgan, J.R.; London, S.; Lyu, M.R. Achieving software quality with testing coverage measures. *Computer* **1994**, *27*, 60–69.
12. Whalen, M.W.; Rajan, A.; Heimdahl, M.P.; Miller, S.P. Coverage metrics for requirements-based testing. In Proceedings of the Proceedings of the 2006 international symposium on Software testing and analysis, 2006, pp. 25–36.
13. Staats, M.; Whalen, M.; Rajan, A.; Heimdahl, M. Coverage metrics for requirements-based testing: Evaluation of effectiveness **2010**.
14. Rajan, A. Coverage metrics to measure adequacy of black-box test suites. In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06). IEEE, 2006, pp. 335–338.
15. Corradini, D.; Zampieri, A.; Pasqua, M.; Ceccato, M. Restats: A test coverage tool for RESTful APIs. In Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2021, pp. 594–598.

16. Grano, G.; Titov, T.V.; Panichella, S.; Gall, H.C. Branch coverage prediction in automated testing. *Journal of Software: Evolution and Process* **2019**, *31*, e2158.
17. Golmohammadi, A.; Zhang, M.; Arcuri, A. Testing RESTful APIs: A Survey. *ACM Trans. Softw. Eng. Methodol.* **2023**, *33*. https://doi.org/10.1145/3617175.
18. Waseem, M.; Liang, P.; Shahin, M.; Di Salle, A.; Márquez, G. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software* **2021**, *182*, 111061.
19. Giamattei, L.; Guerriero, A.; Pietrantuono, R.; Russo, S. Automated Grey-Box Testing of Microservice Architectures. In Proceedings of the 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), 2022, pp. 640–650.
20. Corradini, D.; Zampieri, A.; Pasqua, M.; Ceccato, M. Empirical Comparison of Black-box Test Case Generation Tools for RESTful APIs. In Proceedings of the 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), 2021, pp. 226–236. https://doi.org/10.1109/SCAM52516.2021.00035.
21. Ma, S.P.; Fan, C.Y.; Chuang, Y.; Lee, W.T.; Lee, S.J.; Hsueh, N.L. Using Service Dependency Graph to Analyze and Test Microservices. In Proceedings of the 2018 IEEE 42nd Annual Computer Software and Applications Conference, 2018, Vol. 02, pp. 81–86.
22. Ball, T. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes* **1999**, *24*, 216–234.
23. Villa, O.; Stephenson, M.; Nellans, D.; Keckler, S.W. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In Proceedings of the Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 372–383.
24. Schiewe, M.; Curtis, J.; Bushong, V.; Cerny, T. Advancing Static Code Analysis With Language-Agnostic Component Identification. *IEEE Access* **2022**, *10*, 30743–30761. https://doi.org/10.1109/ACCESS.2022.3160485.
25. Abdelfattah., A.; Schiewe., M.; Curtis., J.; Cerny., T.; Song., E. Towards Security-Aware Microservices: On Extracting Endpoint Data Access Operations to Determine Access Rights. In Proceedings of the Proceedings of the 13th International Conference on Cloud Computing and Services Science - CLOSER. INSTICC, SciTePress, 2023, pp. 15–23. https://doi.org/10.5220/0011707500003488.
26. Zhao, X.; Zhang, Y.; Lion, D.; Ullah, M.F.; Luo, Y.; Yuan, D.; Stumm, M. lprof: A non-intrusive request flow profiler for distributed systems. In Proceedings of the 11th {USENIX} Symposium on Operating Systems Design and Implementation, 2014, pp. 629–644.
27. JavaParser Contributors. JavaParser. https://github.com/javaparser/javaparser, Accessed: 2024.
28. Amazon Web Services. ELK Stack. https://aws.amazon.com/what-is/elk-stack, Accessed: 2024.
29. Elastic. Java High Level REST Client. https://www.elastic.co/guide/en/elasticsearch/client/java-rest/current/java-rest-high.html, Accessed: 2024.
30. Selenium Contributors. Selenium. https://www.selenium.dev, Accessed: 2024.
31. Gatling Contributors. Gatling. https://gatling.io, Accessed: 2024.
32. Zhou, X.; Peng, X.; Xie, T.; Sun, J.; Xu, C.; Ji, C.; Zhao, W. Benchmarking microservice systems for software engineering research. In Proceedings of the Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018; Chaudron, M.; Crnkovic, I.; Chechik, M.; Harman, M., Eds. ACM, 2018, pp. 323–324. https://doi.org/10.1145/3183440.3194991.
33. FudanSELab. Train Ticket Wiki. https://github.com/FudanSELab/train-ticket/wiki, Accessed: 2024.
34. Apache Software Foundation. Apache SkyWalking Documentation. https://skywalking.apache.org/docs, Accessed: 2024.
35. Smith, S.; Robinson, E.; Frederiksen, T.; Stevens, T.; Cerny, T.; Bures, M.; Taibi, D. Benchmarks for End-to-End Microservices Testing, 2023, [arXiv:cs.SE/2306.05895].
36. Cerny, T.; Abdelfattah, A.S.; Maruf, A.A.; Janes, A.; Taibi, D. Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study. *Journal of Systems and Software* **2023**.
37. Wohlin, C.; Runeson, P.; Hst, M.; Ohlsson, M.C.; Regnell, B.; Wessln, A. *Experimentation in Software Engineering*; Springer Publishing Company, 2012.