End-to-End Test Coverage Metrics in Microservice Systems: An Automated Approach

Amr S. Abdelfattah[1], Tomas Cerny[2(✉)], Jorge Yero Salazar[1], Austin Lehman[1], Joshua Hunter[1], Ashley Bickham[1], and Davide Taibi[3]

[1] Computer Science, Baylor University, One Bear Place, Waco, TX 97141, USA
amr_elsayed1@baylor.edu
[2] Systems and Industrial Engineering, University of Arizona, Tucson, AZ, USA
tcerny@arizona.edu
[3] University of Oulu, Oulu, Finland
davide.taibi@oulu.fi

Abstract. Microservice architecture gains momentum by fueling systems with cloud-native benefits, scalability, and decentralized evolution. However, new challenges emerge for end-to-end (E2E) testing. Testers who see the decentralized system through the user interface might assume their tests are comprehensive, covering all middleware endpoints scattered across microservices. However, they do not have instruments to verify such assumptions. This paper introduces test coverage metrics for evaluating the extent of E2E test suite coverage for microservice endpoints. Next, it presents an automated approach to compute these metrics to provide feedback on the completeness of E2E test suites. Furthermore, a visual perspective is provided to highlight test coverage across the system's microservices to guide on gaps in test suites. We implement a proof-of-concept tool and perform a case study on a well-established system benchmark showing it can generate conclusive feedback on test suite coverage over system endpoints.

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

1 Introduction

Microservice architecture enables practitioners to build scalable software systems broken down into a collection of loosely coupled interacting services. Each service is responsible for a specific business capability and can be developed and deployed independently of other services. This allows for faster development cycles, easier maintenance, and better scalability.

However, the end-to-end testing of microservice systems can be challenging due to the system's distributed nature hidden from testers. During E2E system validation, testers primarily interact with the system through its user interface,

© IFIP International Federation for Information Processing 2023
Published by Springer Nature Switzerland AG 2023
G. A. Papadopoulos et al. (Eds.): ESOCC 2023, LNCS 14183, pp. 35–51, 2023.
https://doi.org/10.1007/978-3-031-46235-1_3

thereby concealing the underlying logical system structure. However, microservice architecture entails more intricate details compared to traditional monolithic systems, including multiple services, inter-dependencies, and continuous evolution. Testers may lack knowledge about the specific services being involved and executed within the system. Consequently, they may encounter difficulties in testing all possible scenarios. This complexity introduces challenges in E2E testing of microservice systems, as it obscures crucial details that can influence testing completeness and efficiency.

The extent to which a particular system's microservices are involved in individual E2E tests or E2E test suites should be recognized to give testers better insights into system coverage and test-to-microservice dependencies (i.e., test evolution). E2E tests interact with the system through the user interface which mediates the interaction to microservice endpoint level. Thus, associating tests with impacted microservice endpoints they interact with would provide testers with insights into how comprehensive their test suites are when contrasted to all system endpoints.

This paper aims to establish metrics for calculating the coverage of endpoints in E2E test suites their individual tests, and microservices. Furthermore, it aims to propose a practical method and measurement approach through a case study. This work considers microservice endpoints as the points of overlap between the logical system structure and the E2E tests. It proposes an automated approach mapping individual tests to system microservices and their endpoints to guide testers in test design completeness. With the detailed knowledge of test-to-endpoint associations, testers can better understand their test suite coverage and identify unobvious gaps.

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

- Proposal of three metrics (Microservice endpoint coverage, Test case endpoint coverage, and Complete Test suite endpoint coverage) to assess the coverage of endpoints in E2E testing.
- Metric extraction process and proof-of-concept tool imlementation.
- A practical system case study deriving and validating the coverage metrics.

This paper elaborates on related work in Sect. 2 and describes the metrics and process in Sect. 3. A case study is detailed in Sect. 4 followed by a discussion in Sect. 5 and conclusions in Sect. 6.

2 Related Work

Various studies have identified the lack of assessment techniques for microservice systems. A systematic literature review by Ghani et al. [3] concluded that most articles focused on testing approaches for microservices lacked sufficient assessment and experimentation. Jiang et al. [5] emphasized the need for improved test management in microservice systems to enhance their overall quality.

Waseem et al. [9] 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. [4] 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. [1] 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. [4] 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. [6] 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.

In summary, this paper tackles the gap in assessment techniques for microservice testing. It aims to introduce test coverage metrics and develop an analytical tool that can assess microservice systems and measure their test coverage.

3 The E2E Test Coverage Metrics

This section presents our proposed metrics and provides a comprehensive overview of our automated approach, outlining its stages for extracting the data required for calculating the metrics over systems. The objective is to assess E2E testing suites in achieving coverage of endpoints within microservices-based systems.

3.1 The Proposed Metrics Calculations

E2E testing involves test suites, where 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:

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

$C_{\mathrm{ms}(i)}$- the coverage per microservice i,
$E^{\mathrm{tested}}_{\mathrm{ms}(i)}$ - the set of tested endpoints in microservice i,
$E_{\mathrm{ms}(i)}$ - 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 tests by the total number of endpoints in the system. It provides insights into the completeness of test suites 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.

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 Fig. 1. 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), E2.2 has an inter-service call to endpoint E3.1 in MS-3.

Applying our metrics, 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% ($C_{\text{ms}(2)} = \frac{2}{2}$).

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, which is considered in our approach.

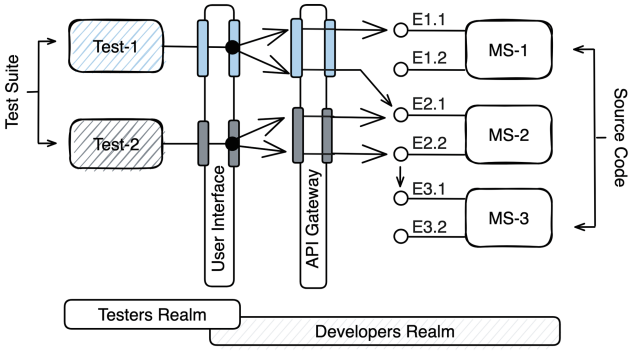


Fig. 1. Calculation Clarification Example

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 ($C_{\text{suite}} = \frac{4}{6}$).

3.2 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 E2E 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 Fig. 2:

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.

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

Stage 1: Expoint Extraction from Source Code (Static Analysis): 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 Fig. 3, 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.

The identification of API endpoints typically relies on specific frameworks or libraries. For example, in the Java Spring framework, annotations such as `@RestController` and `@RequestMapping` are commonly used. 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. [7] or accomplished by frameworks like Swagger[1]

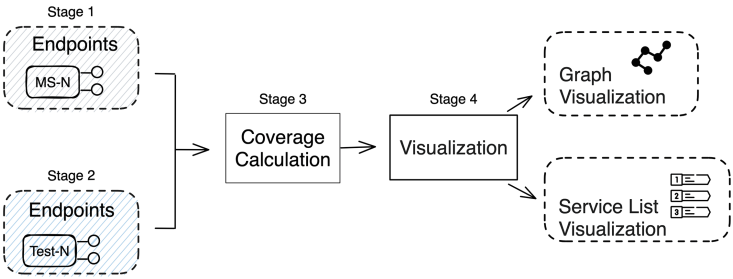


Fig. 2. The proposed approach overview

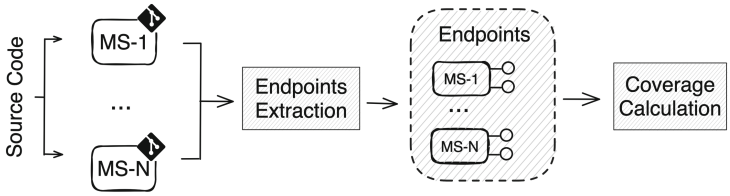


Fig. 3. Stage 1: Static analysis flow

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.

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

Stage 2: Endpoint Extraction from Log Traces (Dynamic Analysis): 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 E2E tests and capturing the traces that occur, as illustrated in Fig. 4.

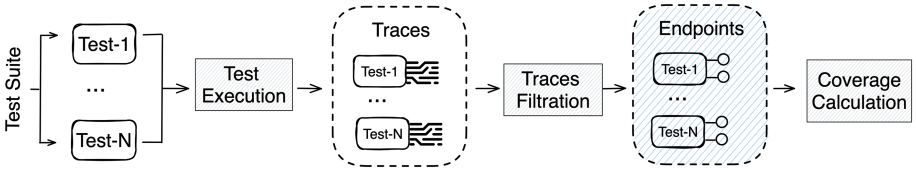


Fig. 4. Stage 2: Dynamic analysis flow

The dynamic analysis flow sketched in Fig. 4 has two main responsibilities. Firstly, it takes the tests and executes them sequentially. During the execution of the E2E 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`[2] 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.

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

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

input for the *coverage calculation process*, where it is combined with the output of the static analysis stage.

Stage 3: Coverage Calculation: This stage combines the extracted equations from the previous two stages to calculate the three metrics of coverage ($C_{\text{ms}(i)}$, $C_{\text{test}(i)}$, C_{suite}).

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, while those identified by static analysis hold parameter types and names. A similar challenge has been accounted for when profiling systems using log analysis and matching log lines with logging statements in the source code [11]. 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. 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 calls 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 effectively exercised by the tests.

Stage 4: Coverage Visualization: The approach offers two ways to visualize 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 Fig. 7a. 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 Fig. 7b. These techniques help in visualizing 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 assess the effectiveness of their testing efforts and identify areas that require improvement.

4 Case Study

To demonstrate the completeness of our approach, we implemented a prototype and conducted a case study on an open-source system benchmark and an E2E test suite designed for the same system. We calculated our metrics on the testbench and compared the results with a manually calculated ground truth.

4.1 Proof of Concept Implementation

This section describes the implementation of a prototype[3] 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[4] library. 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 utilized Apache Maven, a build automation tool for Java projects, to execute our JUnit test suites. JUnit, a widely adopted unit testing framework, offers seamless integration with various automation test frameworks, including Selenium. On the other hand, we focused on extracting logs and traces from Elasticsearch, which is widely adopted as a central component in the ELK[5] (Elasticsearch, Logstash, Kibana) stack. We used the Elasticsearch Java High-Level REST Client[6], 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.

Then, the prototype performs the coverage calculation (Stage 3). It integrates the results of the static and dynamic processes, and applies the proposed metrics. For the coverage visualization (Stage 4), we provided the two visualization approaches discussed earlier. We implemented a web application[7] 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[8], an open-source project that generates the graph from source code. Additionally, we

[3] Prototype: https://github.com/cloudhubs/test-coverage-backend.
[4] JavaParser: https://github.com/javaparser/javaparser.
[5] ELK: https://aws.amazon.com/what-is/elk-stack.
[6] Elasticsearch Java Client: https://www.elastic.co/guide/en/elasticsearch/client/java-rest/current/java-rest-high.html.
[7] Coverage Visualizer: https://github.com/cloudhubs/test-coverage-frontend.
[8] Prophet: https://github.com/cloudhubs/graal-prophet-utils.

utilized the visualizer library[9], which offers a tailored 3D microservices visualization for service dependency graphs.

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 [2] 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[10] 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[11] 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 prototype to utilize SkyWalking for forwarding system logs to Elasticsearch for additional processing and analysis.

For the test suites, we utilized an open-source test benchmark[12] published in [8]. This benchmark aims to test the same version of the TrainTicket system. It contains 11 E2E test cases using the Selenium 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 an open accessed dataset[13]. This involved manual extraction of the data related to the first two stages in our proposed process in Sect. 3.2, 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 validated the endpoints extracted during the dynamic analysis by examining the Selenium test suites. Since the Selenium tests do not explicitly reference endpoints but rather perform UI-based actions, we manually analyzed the logs generated by the tests, which were stored in Elasticsearch. These logs contained encoded information about the source and destination endpoints,

[9] 3D Visualizer: https://github.com/cloudhubs/graal_mvp.
[10] SkyWalking: https://skywalking.apache.org/docs.
[11] TrainTicket: https://github.com/cloudhubs/train-ticket/tree/v1.0.1.
[12] Test benchmark: https://github.com/cloudhubs/microservice-tests.
[13] Dataset: https://zenodo.org/record/8055457.

which we decoded and filtered to extract the list of endpoints called during the tests. It produced 171 unique endpoints from the logs.

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 prototype executed the test cases from the provided test benchmark, generated the list of called endpoints and calculated the test coverage according to the described metrics.

The results of the experiment execution revealed a total of 171 unique endpoints extracted from a set of 953 log records generated during the execution of the test cases, out of which 119 endpoints are actual endpoints within the system, 52 endpoints that are related to API-gateway calls. The complete data analysis phases with their results are published in a dataset (see footnote 13). This dataset contains the complete calculations of $C_{\text{ms}(i)}$, $C_{\text{test}(i)}$ metrics.

In terms of evaluating the completeness of our prototype, this case study confirmed that we captured all the endpoints declared in the ground truth. The prototype successfully captured all 262 implemented endpoints in the system, demonstrating the completeness of Stage 1 outcome. For Stage 2 completeness, the prototype extracted all 171 endpoints. Out of the total 171 endpoint calls, our prototype identified 52 distinct calls associated with the API gateway, which are not considered actual endpoints in the system.

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 5 illustrates the endpoint coverage achieved by the 11 test cases, along with the average coverage for better measurement.

Table 1. Summary Statistics of Coverage Metrics

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

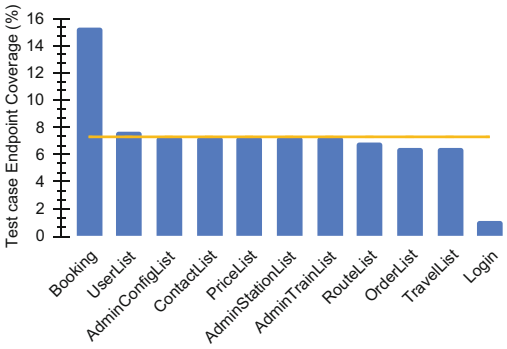


Fig. 5. Test case Endpoint Coverage in the Benchmark Test cases ($C_{\text{test}(i)}$)

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 the test suite completely missed testing any endpoints in the following four microservices: *ts-wait-order-service, ts-preserve-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 Fig. 6.

The metrics calculations are visualized using two visualization approaches, as shown in Fig. 7a and Fig. 7b. One with per service view and the other providing the holistic service dependency overview in the context of endpoint coverage. For example, the `ts-config-service` microservice has an approximate coverage of 83.33%, missing only one out of six endpoints. This information is also represented in yellow color in the 3D graph visualization, where the color of each node corresponds to the coverage percentage of the respective microservice.

5 Discussion

Our approach has shown promising results in mitigating E2E test degradation and contributing to the continuous reliability and quality assurance of decentralized microservice systems. While further comprehensive data analysis is ongoing, initial findings indicate a positive impact. It determines the log traces connecting tests with endpoints from the current system and a current test suite by automated means. Such traces can help testers manage change propagation as it directly indicates a co-change dependency between specific microservices or endpoints and particular tests. Furthermore, integrating it with CI/CD pipelines would make it an ideal tool to ensure coverage across system evolution changes. On the other hand, it is crucial to consider the context in which the approach

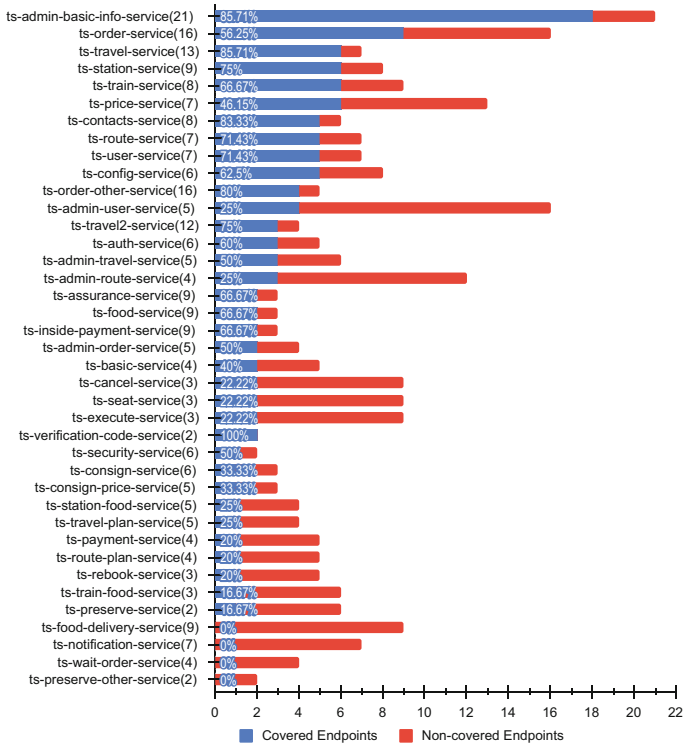


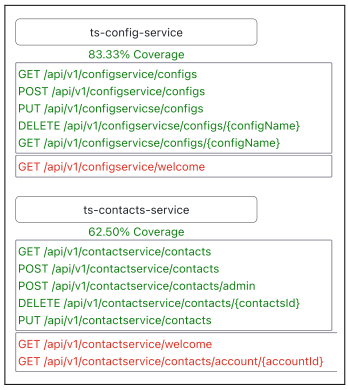
Fig. 6. Microservice Endpoint Coverage in the Benchmark System ($C_{\text{ms}(i)}$) *The numbers in parentheses indicate the total number of endpoints in each ms.*

is applied, as the user interface 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. At the same time, it raises the question of whether the remaining endpoints represent the smell known as *Nobody Home* where the wiring is missing from the user interface, or possibly the endpoints are outdated or dead code.

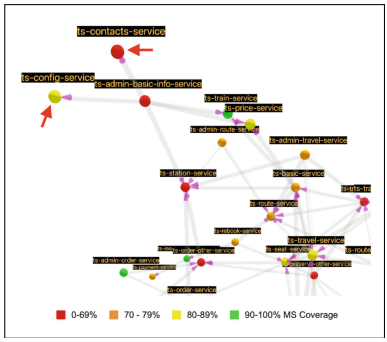
It is worth noting that microservices often implement `isAlive` endpoints for health checks. While some libraries, like Hystrix, can automatically generate these endpoints, some systems implement them manually. As an example, Train-Ticket implemented 39 endpoints that were not utilized in the user interface, rendering them meaningless. Nevertheless, validating these endpoints can guarantee that the system is correctly initialized.

5.1 Threats to Validity

In this section, we address the potential validity threats to our approach. We adopt Wohlin's taxonomy [10], which encompasses construction, external, internal, and conclusion threats to validity, as a framework for our analysis.



(a) Microservice endpoint list



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

Fig. 7. Microservices endpoint coverage visualization (full pictures (see footnote 7))

A potential **construction validity threat** arises from the dependency on static analysis for endpoint extraction and dynamic analysis of centralized traces generated by E2E tests. It includes missing or non-standard source code and a lack of support for centralized traces, which can hinder our approach.

Our prototype 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.

6 Conclusion

Despite the broad adoption of microservices for software solutions, there are open challenges practitioners face with E2E testing. While testers might assume complete test coverage, verification mechanisms on the actual state of test completeness within the system are missing. We sought to define metrics and establish an approach to calculate the E2E test suites coverage of microservice system endpoints. Our approach determines the connection between individual tests and microservice endpoints, which are the system entry points for user interfaces used by E2E testers. We performed a case study on an established system benchmark and a test suite aiming for full coverage, revealing that the achieved coverage fell significantly short of being comprehensive.

In future work, we will explore system and test suite evolution, evaluating how our approach guides co-coupling between system changes and tests to ensure quality assurance and reduce test suite degradation. We also plan to expand our metrics to encompass different test paths within the endpoints.

Acknowledgements. This material is supported by the National Science Foundation under Grant No. 2245287 and Grant No. 349488 (MuFAno) from the Academy of Finland.

References

1. Corradini, D., Zampieri, A., Pasqua, M., Ceccato, M.: Empirical comparison of black-box test case generation tools for restful APIs. In: 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 226–236 (2021). https://doi.org/10.1109/SCAM52516.2021.00035
2. FudanSELab: Home. https://github.com/FudanSELab/train-ticket/wiki
3. Ghani, I., Wan-Kadir, W.M., Mustafa, A., Imran Babir, M.: Microservice testing approaches: a systematic literature review. Int. J. Integr. Eng. **11**(8), 65–80 (2019). https://publisher.uthm.edu.my/ojs/index.php/ijie/article/view/3856
4. Giamattei, L., Guerriero, A., Pietrantuono, R., Russo, S.: Automated grey-box testing of microservice architectures. In: 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), pp. 640–650 (2022)
5. Jiang, P., Shen, Y., Dai, Y.: Efficient software test management system based on microservice architecture. In: 2022 IEEE 10th Joint International Information Technology and Artificial Intelligence Conference, vol. 10, pp. 2339–2343 (2022)
6. 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: 2018 IEEE 42nd Annual Computer Software and Applications Conference, vol. 2, pp. 81–86 (2018)

7. Schiewe, M., Curtis, J., Bushong, V., Cerny, T.: Advancing static code analysis with language-agnostic component identification. IEEE Access **10**, 30743–30761 (2022). https://doi.org/10.1109/ACCESS.2022.3160485
8. Smith, S., et al.: Benchmarks for end-to-end microservices testing (2023)
9. Waseem, M., Liang, P., Shahin, M., Di Salle, A., Márquez, G.: Design, monitoring, and testing of microservices systems: the practitioners' perspective. J. Syst. Softw. **182**, 111061 (2021)
10. Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A.: Experimentation in Software Engineering. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29044-2
11. Zhao, X., et al.: lprof: a non-intrusive request flow profiler for distributed systems. In: 11th {USENIX} Symposium on Operating Systems Design and Implementation, pp. 629–644 (2014)