Benchmarking Component and Integration Testing in Microservices: Test Suites and Fault Analysis on TrainTicket

Lena Gregor
School of CIT
Technical Univerity of Munich
Munich, Germany
lena.gregor@tum.de

Marcel Skalski
School of CIT
Technical University of Munich
Munich, Germany
marcel.skalski@tum.de

Alexander Pretschner
School of CIT
Technical University of Munich
Munich, Germany
alexander.pretschner@tum.de

***Abstract*—Microservices decompose systems into small, independent services, offering benefits that have led to widespread adoption. However, their distributed nature, diverse technology stacks, and independent development introduce significant challenges in ensuring system reliability. While numerous automated testing approaches have been proposed, only a few manually implemented test suites are openly available for comparison. To the best of our knowledge, no extensive benchmark test suites for component and integration testing of microservices are publicly accessible. This work addresses this gap by providing two benchmark test suites — one on the component and one on the integration level — developed within a case study on the well-established TrainTicket microservice system. The test suites were initially implemented by four independent teams and later unified by the authors, yielding 1,365 component and 210 integration tests in total. They uncover 51 faults, 8 untypical design decisions, and 1 fallback-related issue within the system. We further manually analyze the detected faults and compare them with existing fault taxonomies, identifying 11 gaps in current classifications. Additionally, we provide insights into the relation between the test case type and fault type it detected. Our insights can support future evaluations of new approaches and foster the development of more comprehensive fault models for microservice systems.**

***Index Terms*—fault dataset, microservices testing, integration tests, component tests, benchmark test suites**

I. Introduction

The microservice architectural style is an approach to software development where an application is composed of multiple small, independent services. Each service operates in its own process and communicates with others using lightweight protocols such as REST APIs or message queues. The services are designed around specific business capabilities and can be deployed independently. Development and management are decentralized, enabling flexibility in selecting programming languages and data storage solutions for each service. [1], [2]

Because of these benefits, microservice architecture has seen widespread adoption in industry [3], [4]. However, the distributed nature of microservices, combined with the variety of technology stacks across services, introduce significant complexities not only in building reliable systems but also in testing—challenges that often exceed those found in monolithic architectures [5]. To help facilitate and improve the testing of microservice systems, many automated approaches are developed and published in the literature [6]–[11]. Existing automated testing approaches primarily focus on isolated components or entire system-level testing, often neglecting the intermediate integration testing, which is critical for validating service interactions. To the best of our knowledge, they are usually evaluated based on failure rate and against other existing automatic approaches but not against manually written test cases (e.g. [6]) due to the lack of extensive test suites in open-source systems. To facilitate the evaluation of new (automated) approaches, Smith et al. [12] created manually written benchmark test suites on the E2E-test level for two existing microservice systems. However, to the best of our knowledge, there are no openly available and extensive test suites on the component and integration level. To address this gap, we construct new component- and integration-level test suites through a case study on TrainTicket [13]—a benchmark microservice system designed for railway ticket booking, which has been widely adopted in previous research [6], [13]–[21]. To validate the effectiveness of those test suites, we report and analyze the 51 faults that we found through testing the system. Additionally, we compare those faults with existing knowledge about faults in microservice systems by classifying them into three existing fault and issue taxonomies [22]–[24] and one more general grouping [13]. This classification provides a structured understanding of the faults that exist in the TrainTicket system. By identifying gaps in existing taxonomies, our study highlights fault types that are not yet well-documented, which can inform future automated testing approaches and microservice fault mitigation strategies. The contributions of this paper are as follows:

- **Extensive Test Suites:** Test suites with 1,365 component and 210 integration tests open-source available.
- **Labeled Fault Dataset:** A dataset of 51 detected and manually classified faults within the benchmark system.
- **Identification of New Fault Categories:** Elicitation of

11 fault categories that were not documented in the microservice literature before.
- **Mapping of Fault Types to Test Cases:** Analysis of fault types and the specific test cases that detected them, offering insights into test effectiveness.
- **Methodology for Analyzing Fault Resolution:** A structured approach to identifying whether and how detected faults were addressed in subsequent development, including the interpretation of code changes, demonstrated through a case study.

The work presented in this paper enables several important directions for future work, which are addressed in the following.

Evaluation of New Test Quality Criteria. Our benchmark test suites provide a foundation for assessing new test quality criteria at the component and integration levels in microservice systems. Similar to how Smith et al. [12] enabled the evaluation of microservice-specific coverage metrics for E2E testing [25], our test suites allow for similar evaluations at finer granularities. Since no extensive test suites for these levels are openly available, this resource can drive new research in test quality.

Evaluation of New Test Generation Approaches. The TrainTicket fault dataset enables a more detailed evaluation of new test generation approaches for microservice systems and REST APIs. While existing approaches are so far assessed by failure rates, our dataset allows for deeper analysis, including undetected faults, improving test effectiveness insights.

Expanding the Fault Dataset for Statistical and Machine-Learning-based Analysis. Our labeled dataset of 51 detected faults can be expanded by analyzing additional microservice systems. This would allow researchers to study fault distribution patterns across different architectures, industries, or service compositions. A larger dataset could also enable machine learning models to predict potential faults in microservice systems based on architectural or operational characteristics.

Refining Microservice Fault Taxonomies. Our identification of new fault types suggests gaps in existing taxonomies, for which we have proposed new fault categories. Future research could refine those fault classifications for microservices by conducting systematic studies across multiple benchmark systems to determine whether our new fault types generalize. This could lead to an improvement of the existing microservice fault taxonomies, which would help researchers and practitioners to better structure their fault detection and mitigation strategies.

II. Background

A. Test Level in Microservice Systems

Traditionally, there are three test levels, namely unit, integration, and system level testing. In the context of microservices, we can find additional test levels, such as component and contract testing in the literature [26] and practice. In this paper, we adopt the following definitions of test levels:

a) Unit Testing: Testing which focuses on the internals of a single microservice, e.g., an algorithm within the domain logic, the correct functioning, and communication between the layers or modules inside the microservice.

b) Component Testing: Testing a single microservice through its interface in isolation. If the service is dependent on any other services, those services need to be mocked.

c) Integration Testing: Testing the connection between two or more microservices within a system. Further connected microservices can be mocked and iteratively added to the subset that is tested through integration strategies such as bottom-up or top-down integration.

d) System Testing: Testing the complete microservice backend through the endpoints provided to the client.

e) E2E Testing: Testing the complete system through the user interface or client interface.

B. Functional Testing

Functional testing is a software testing approach that evaluates whether a system or component behaves as expected by verifying its functionality against specified requirements. It focuses on testing inputs and expected outputs without considering the internal structure of the system. [27]

C. Defects

The terms failure, error, and fault are often inconsistently used in research and in practice [28]. Within this work, we adopt the three definitions proposed by Avizienis et al. [29]:

- *Failure:* A noticeable deviation between the intended behavior and the actual behavior of an executed system. Every failure stems from an error.
- *Error:* The intended state of the system deviates from its actual state. An error may or may not cause a failure but always stems from a fault.
- *Fault:* The actual or hypothesized reason for the deviation, usually a bug in the implementation. A fault can also exist within inactive code segments and, therefore, does not necessarily trigger an error.

Similar to Pretschner [30], we use *defect* as an umbrella term for failure, error, and fault.

III. Case Study

We follow the guidelines of Runeson and Höst [31] for designing and conducting case studies and first define the *objectives* in the form of research questions. We then split the case study and address the research questions individually, where we report on the study design, procedure, and results.

A. Research Questions

Within our case study, we want to address the following research questions:

RQ1: What does a unified benchmark test suite for component and integration testing look like in terms of its composition, fault-detecting capabilities, and types of test cases?

RQ2: Which faults were detected through testing in the benchmark system?

RQ3: How do the faults identified in the benchmark system compare to those reported in related work?

RQ4: Which faults were fixed in subsequent development as indicated by the commit history?

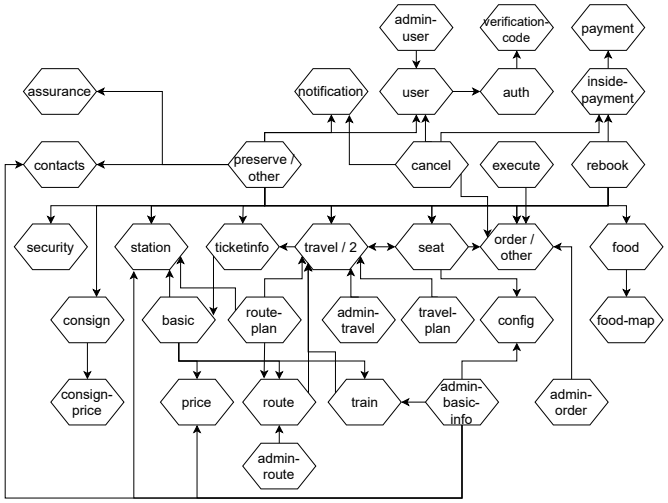

Fig. 1. Architecture of TrainTicket based on the original architecture diagram provided in the TrainTicket repository[2].

RQ5: Which kind of test cases helped to detect which types of faults?

B. Study Object

To conduct our case study on component and integration level test suites, we use a well-established open-source microservice system as the study object. The TrainTicket system was developed to provide a benchmark microservice system [13] and is, therefore, often used for research [6], [13]–[21]. The system provides a train ticket booking process, including registering and logging in users, searching for, paying for, reserving, and modifying tickets, and placing food orders. Smith et al. [12] created a benchmark test suite on the E2E-test level and Giametti et al. [6] used test suite generation tools to create test cases on the system test level for the TrainTicket system. However, to the best of our knowledge, there are currently no human-written, extensive test suites on the component or integration level publicly available for this system. The TrainTicket system consists of 41 services (as of version `0.0.4`). Figure 1 shows an overview of the system architecture. We decided to use version `0.0.4` in our experiments as this version relies on a Docker environment. We noticed from related work, that many test case generation frameworks and other testing tools rely on Docker for deployment, which makes our test suites compatible with as many research tools as possible later. Additionally, testing an older system version will give us the ability to analyze the bugs we find in the more recent system version and show potential fixes for those faults.

C. RQ1: Component and Integration Test Suites

1) Study Design: The goal of this part of the case study is to create benchmark test suites at the component and integration level. To achieve this, we employ a structured approach in which four independent student teams develop test suites following predefined test selection principles for functional testing. Each team creates two separate test suites (one for component testing and one for integration testing) ensuring a diverse and systematic exploration of the system under test. Afterward, we consolidate those test suites into two unified benchmark test suites, one at the component level and one at the integration level.

This approach is chosen for several reasons. First, by involving multiple independent teams, we mitigate the risk of bias from any single perspective and ensure a broad and varied set of test cases. Each team applies two structured test selection techniques: equivalence class-based selection [32] and defect-based selection [30], [32]. The equivalence class-based approach systematically covers input variations, while the defect-based approach focuses on common fault patterns. This structured yet flexible methodology ensures methodological rigor while allowing each team to bring in diverse testing perspectives. The final consolidation step, where we combine the individually developed test suites into two comprehensive benchmark suites, further enhances the readability and completeness of the final artifacts.

2) Study Procedure: We recruited 15 students from the faculty of computer science who were all at least in their third year of studying for their computer science degree. The study was conducted in the form of a practical course. The students had to apply with their CV, grade transcript, and motivational statement which we used as a basis to choose suitable candidates with good grades and suitable preliminary experience in software development and testing. Prior to the main project, the students participated in four teaching sessions on the principles of microservice development and general testing practices, including unit testing, component testing, and integration testing. Additionally, the students had to fulfill coding assignments, which included practical training and assessments of their microservice development and testing skills. Then, the students were distributed into four teams and each got the same task: to design and implement test suites on the component and integration test level for the TrainTicket system over a period of three months.

Before designing the test suites, the students adjusted the system's original configuration to ensure it was executable for all participants. These adjustments involved updating existing dependencies and removing deprecated ones. To ensure that the test suites could later be combined into two unified test suites, the students agreed on an executable configuration and a suitable test tech stack. This tech stack included frameworks such as JUnit 5[3], Testcontainers[4], WireMock[5], and Mockito[6].

Because the system lacks official specifications, the students first had to write their own specifications to be able to develop functional test suites later. To do this, they had to infer the intended behavior of the individual services from the existing code, especially names of endpoints and parameters, and documentation. Where possible, they used standard best practices for microservice systems or REST APIs as the intended behavior, e.g., the correct usage of HTTP/REST

[2] https://github.com/FudanSELab/train-ticket

[3] https://junit.org/junit5/ [4] https://testcontainers.com/
[5] https://wiremock.org/ [6] https://site.mockito.org/

methods [33]. Whenever a specific pattern was used throughout most services, it was considered as intended behavior even if it did not adhere to common best practices. For example, the TrainTicket system consistently does not adhere to the REST API response code best practices [34]. Most services always return a 200 status code (which is meant for correct responses, where everything went well [34]) but then include a status field in the response body, where 1 indicates that everything went well and 0 indicates that there was an error. As this pattern was used in most services, we infer that it was intended by the developers and wrote the specifications accordingly. Once the specifications were established, the teams proceeded to develop test suites at the component and integration levels.

The students used two established strategies for test case selection: the equivalence-class-based method [32] and the defect-based method [30], [32]. The equivalence-class-based technique helps ensure that test cases represent a diverse range of inputs by grouping them into equivalent classes, where each class is expected to produce similar outcomes [32]. For example, when testing a payment service, equivalence classes could be defined for valid and invalid credit card numbers, different amounts, or varying levels of account balance. For the defect-based method, the students used established defect and fault taxonomies [35]–[37] to create their fault hypotheses.

All services were first tested in isolation on the component level, mocking all dependent services. The students then used the bottom-up integration strategy for the integration test layer and designed test cases for each individual integration layer.

After this phase, we merged the individual test suites into two cohesive test suites: one for the component and one for the integration level. This consolidation process involved selecting unique and effective test cases (those that were successful in finding defects) while eliminating redundant tests. Our consolidation strategy comprised three key steps:

1) *Initial Selection:* We began by adopting the test cases from the team with the highest number of tests, assuming that this team likely covered the widest range of unique scenarios.
2) *Verification:* We executed and debugged each test case from this initial set locally to ensure their correctness. Tests that failed due to misconfigurations or unreported errors were either adjusted accordingly or excluded from the consolidated test suite.
3) *Supplementary Selection:* We then reviewed the test cases from the remaining teams, incorporating only those that provided unique coverage.
4) *Supplementary Inclusion:* To ensure comprehensive coverage of common scenarios in the system, we identified test cases that could be universally applied to various services and replicated them accordingly. For instance, test cases verifying the behavior of endpoints when provided with malformed request bodies were extended to all services with endpoints that accept request bodies, even if no team had originally written such tests for a particular service. Similarly, if we discovered faults not explicitly addressed by any team's test cases while combining the test suites, we implemented additional test cases to explicitly target these faults.

We adjusted the source code of the TrainTicket system with careful consideration to preserve the original behavior of the services under two conditions:

1) Need for Additional Data on Service Startup: Some tests required specific database states not initially provided by the system. To enable such scenarios, we added extra entities to the database initialization without modifying or removing existing data.
2) Resolution of Testing Blockers: Faults that prevented integration tests from executing were resolved only as a last resort when no viable workarounds existed.

To improve organization and navigation, test cases were categorized into component and integration test directories, reflecting their respective test levels. Within these directories, they were further grouped by HTTP methods (e.g., `GET`, `POST`, `PUT`, `DELETE`), providing an overview of the endpoints offered by each service and simplifying searches for specific test cases. All test cases follow the Arrange-Act-Assert (AAA) pattern, a pattern for structuring test cases [38], [39].

Finally, we introduced specific prefixes to help identify test cases that uncovered faults. Test cases that failed due to a fault in the system were prefixed with `FAILING`. Those related to unreliable endpoints, such as ones using random functions, were labeled `UNSTABLE` to indicate that failures occurred depending on the function's outcome. Test cases that originally failed but were later resolved in the source code were prefixed as `FIXED_PREVIOUSLY_FAILING` to indicate that they would have failed if the fault had not been resolved.

3) Study Results: Team 1 created a total of 1160 component test cases and 200 integration test cases, Team 2 developed 392 component and 75 integration test cases, Team 3 implemented 1427 component and 515 integration test cases, and Team 4 created 602 component and 234 integration test cases. The four teams employed different strategies for creating equivalence class-based and defect-based tests cases. For instance, some teams developed frameworks to generate equivalence classes to aid in the test case creation process, while others manually selected appropriate equivalence classes. For defect-based testing, some teams focused on using boundary values for their inputs, while others manipulated the environment, e.g., by setting the database to an atypical state, such as leaving it empty or filling it with unexpected data. Additionally, certain teams prioritized security-related test cases, whereas others focused more on verifying the logical correctness of services.

The final test suites consist of 1,365 component and 210 integration test cases across 37 and 19 services, respectively, with between 7 and 134 test cases per service, which are provided in a GitHub repository [40]. A summarized breakdown of the number of test cases per service is provided in the replication package [40]. It is worth mentioning that the consolidated test suites contain fewer test cases than some of the individual team test suites due to the removal of redundant test cases. We systematically identified and excluded duplicate

tests, to ensure that the final test suite remains comprehensive while minimizing redundancy.

Creating the combined test suite revealed that, for the TrainTicket system, each test for an endpoint falls into one of three categories: *Happy Path Tests* [41], [42], *Edge Case Tests*, or *Syntactic Tests*.

- *Happy Path Tests* verify that a service behaves as expected under normal operating conditions. These test cases represent scenarios where the service is expected to return the intended output for a typical set of input values.
- *Edge Case Tests* explore scenarios with valid but unusual inputs or environment settings that may lead to exceptions due to unanticipated conditions in the service logic.
- *Syntactic Tests* focus on manipulating the format or structure of input data without requiring knowledge about the specific semantics of the service. These test cases assess the service's ability to handle malformed inputs, such as malformed path variables or request bodies.

The concrete implementation for each test case type depends on the HTTP method of the endpoint under test, the kind of input (path variable or request body), and the testing level (component or integration test suite). Typically, each test class in our component test suite includes at least one test case for each type, whereas our integration test suite contains only *Happy Path Tests* and *Edge Case Tests*. Across both test suites, 130 test cases were labeled as `FAILING`, 24 as `FIXED_PREVIOUSLY_FAILING`, and 1 as `UNSTABLE`, demonstrating their ability to detect faults.

D. RQ2: Detected Faults

1) Study Design: The four student teams reported the defects they found during testing the TrainTicket system. To collect those mentions of defects, the reports and test case implementations from the student teams were reviewed using a keyword-based search for terms like "fail", "error", and "bug". Identifying faults from the list of mentioned defects involved setting breakpoints at the faulty part of code described by the students and executing their test cases in the debugger. If we observed a deviation between the system's actual state and its intended state after evaluating the described part in the source code, we identified the issue as a fault. In some cases, the reports contained descriptions of errors or failures without identifying their underlying faults. For instance, one student noted that a *"user can be created with an empty password and its length can be less than 6 characters."* in the `ts-auth-service`. While this described a behavioral deviation (the service should reject passwords with a length shorter than six characters), it did not specify the underlying fault. In such cases, we manually identified the fault causing the failure by reviewing the test case to ensure its correctness. If the test case contained no issues, we closely reviewed the implementation of the source code and executed the test case using a debugger to pinpoint the fault.

2) Study Procedure: Due to the lack of formal specifications, determining whether a reported issue was a fault was often difficult. While specifications were created during the course, they were derived from the source code, potentially embedding incorrect behaviors as intended ones. The absence of original specifications further complicated fault identification, requiring inferred interpretations of intended behavior.

For example, the `ts-cancel-service` proceeds with cancellations regardless of whether the `ts-inside-payment-service` successfully processes refunds. Without a specification, it remains unclear whether this behavior is faulty or intentional. To address such ambiguities, four classifications were introduced:

- *Fault:* A clear deviation between actual and intended (but inferred) behavior.
- *Untypical Design Decision:* No definite fault, but an unconventional design choice that may cause future problems, e.g., dead code or redundant services.
- *Non-Fault:* The issue stemmed from a test case error, misconfiguration, or a reasonable alternative interpretation of expected behavior.
- *Special Fallback Issue:* Ambiguities related to how fallback mechanisms handle failures, with some teams considering them faults and others valid exception handling.

To facilitate efficient referencing to these classified issues in subsequent analysis, we introduced a naming convention and applied it to the categories of Fault, Untypical Design Decision, and Special Fallback Issue. Faults were abbreviated with an "F" followed by a number (e.g., F1, F2, F3) to indicate unique fault types. If the same fault type occurred across multiple services, an additional letter was appended to Fx to reference the respective service. For instance, F5a might refer to a fault in the `ts-train-service`, while F5b would indicate the same fault type in the `ts-station-service`. To avoid inflating the fault count, we did not count multiple occurrences of the same fault type within a single service. Nonetheless, we documented these occurrences in the replication package [40]. Untypical Design Decisions were similarly labeled with a "D" followed by a number (e.g., D1, D2). Issues related to the unique category of Special Fallback Issue were collectively referred to as "S1".

3) Study Results: We initially identified a total of 147 issues through the reports and comments of test implementations. Of these, 78 described faults, 15 referred to untypical design decisions, 22 were discarded as non-faults, and 32 were labeled as special fallback issues. Since different teams occasionally reported the same fault or design decision under different descriptions, we consolidated overlapping reports by grouping multiple mentions of the same issue. After this consolidation process, we identified a total of 51 faults and 8 untypical design decisions. However, many of these 51 faults were repeated occurrences of the same fault type across multiple services. Using our naming convention, we assigned the same number to identical fault types across services, differentiating their occurrences with letter suffixes. When considering the faults without letter suffixes, we arrive at 27 distinct fault types. All detected faults are reported in Table I and all untypical design decisions in Table II. An excerpt of those is explained in more detail below. For detailed descriptions of all issues, please refer to the replication package [40].

a) F11: The fallback method in the controller class of the `ts-user-service` returns `HTTPEntity`, while other methods

TABLE I
FAULTS IDENTIFIED IN THE TRAINTICKET SYSTEM AND THEIR CATEGORIZATION INTO TAXONOMIES OF WASEEM ET AL. [22] AND SILVA ET AL. [23]
TYPES OF ISSUES AND FAULTS IN BOLD REPRESENT OUR NEWLY PROPOSED CATEGORIES FOR THOSE TAXONOMIES

Detected Faults		**Taxonomy Classification by Waseem et al.**			**Taxonomy Classification by Silva et al.**		
Fault ID	**Fault Description**	**Category**	**Sub-Category**	**Type of Issue**	**Sub-Category**	**Fault**	**Characteristic**
F1	Wrong status codes in HTTP responses	Technical Debt	Code Debt	Inconsistent Code	Analysability	**Invalid Response Data Fault**	Maintainability
F2	Incorrect password-length validation	Technical Debt	Code Debt	**Logic Sequence Error**	Confidentiality	**Insufficient Credentials Accepted**	Security
F3	Overwriting instead of combining results	Technical Debt	Code Debt	**Data Overwrite Error**	Functional Completeness	Internal Fault	Implementation
F4	Incorrect `null`-checks on collections	Technical Debt	Code Debt	**Condition Logic Error**	Functional Completeness	Internal Fault	Implementation
F5	Incorrect `null`-checks on `Optional` objects	Technical Debt	Code Debt	**Condition Logic Error**	Functional Completeness	Internal Fault	Implementation
F6	Incorrect handling of non-existent routes	Technical Debt	Code Debt	**Condition Logic Error**	Functional Completeness	Internal Fault	Implementation
F7	Index conflict with `ConsignPrice` objects in database	Technical Debt	Code Debt	**Insufficient Value Validation**	Integrity	Not Validating Input/Data	Security
F8	Missing null-checks leading to an unhandled `NullPointerException`	Technical Debt	Code Debt	**Insufficient Value Validation**	Functional Completeness	Internal Fault	Implementation
F9	Improper date handling in URL path variables	Technical Debt	Code Debt	**Insufficient Value Validation**	Functional Completeness	Internal Fault	Implementation
F10	Misuse of `java.util.Date` constructor	Technical Debt	Code Debt	**Initialization Error**	Functional Completeness	Internal Fault	Implementation
F11	Inconsistent return types in fallback methods	Compilation Issue	Syntax Error		Analysability	**Invalid Response Data Fault**	Maintainability
F12	Publicly accessible admin paths due to improper security configuration	Security Issue	Authorization	Handling Authorization Header	Authenticity	**Faulty Authorization**	Security
F13	Invalid endpoint call for empty path variables	Technical Debt	Code Debt	**Insufficient Value Validation**	Analysability	Invalid Request Data Fault	Maintainability
F14	Incorrect usage of `BigDecimal.add`	Technical Debt	Code Debt	**Ignored Return Value**	Functional Completeness	Internal Fault	Implementation
F15	Improper ordering of security matchers	Security Issue	Authorization	Handling Authorization Header	Authenticity	**Faulty Authorization**	Security
F16	Insufficient timeout settings	Exception Handling	Communicatic Exception	Timeout Error	Temporal Behavior	Long Response Time	Service Discovery
F17	Misconfigured initialization data	Technical Debt	Code Debt	**Insufficient Value Validation**	Integrity	Not Validating Input/Data	Security
F18	Comparing `travelDate` to the wrong field	Technical Debt	Code Debt	**Condition Logic Error**	Functional Completeness	Internal Fault	Implementation
F19	Reversed logic for cookie validation	Technical Debt	Code Debt	**Condition Logic Error**	Functional Completeness	Internal Fault	Implementation
F20	Missing `isEmpty`-check on response data	Technical Debt	Code Debt	**Insufficient Value Validation**	Integrity	Not Validating Input/Data	Security
F21	Potentially infinite loop in seat assignment	Technical Debt	Code Debt	**Infinite Loop**	Functional Completeness	Internal Fault	Implementation
F22	Incorrect status code for missing authorization	Technical Debt	Code Debt	Inconsistent Code	Analysability	**Invalid Response Data Fault**	Maintainability
F23	Wrong order of `if`-conditions	Technical Debt	Code Debt	**Logic Sequence Error**	Functional Completeness	Internal Fault	Implementation
F24	Incorrect HTTP method in HTTP request	Service Execution	Service Communication	HTTP Connection	Analysability	Invalid Request Data Fault	Maintainability
F25	Comparing enumeration constant to the wrong field	Technical Debt	Code Debt	**Condition Logic Error**	Functional Completeness	Internal Fault	Implementation
F26	Security configuration mismatch	Security Issue	Authorization	Handling Authorization Header	Authenticity	**Faulty Authorization**	Security
F27	Missing check for `UUID` in path variable	Technical Debt	Code Debt	**Insufficient Value Validation**	Integrity	Not Validating Input/Data	Security

in the same context return `ResponseEntity<Response>`. This mismatch causes a compile-time error when the service is executed.

b) F12: The restriction of access to specific endpoints was not configured to account for the presence of trailing slashes at the end of the path. As a result, an endpoint restricted to administrators (e.g., `POST /api/v1/orderservice/order/admin`) becomes publicly accessible without requiring authorization by appending a trailing slash to the URL (`POST /api/v1/orderservice/order/admin/`)

TABLE II
UNTYPICAL DESIGN DECISIONS IDENTIFIED IN TRAINTICKET

ID	Description
D1	Deleting object from database before validating its existence
D2	Misleading non-matching results
D3	Suboptimal user identification
D4	Redundant input parameters
D5	Duplicated services
D6	Ineffective check for duplicated IDs
D7	Improper delete method design
D8	Redundant service design

TABLE III
CATEGORIZATION INTO THE INTEGRATION FAULT TAXONOMY BY GREGOR ET AL. [24]

Fault ID	Category	Sub-Category	Sub-Sub-Category
F12	Connection Fault	**Unauthorized Access Granted**	
F13	Execution Fault	Incorrect Result	Incorrect Input
F15	Connection Fault	**Unauthorized Access Granted**	
F16	Execution Fault	Timed Out	Service Too Slow
F24	Connection Fault	Connection Denied	Wrong Configuration
F26	Connection Fault	**Unauthorized Access Granted**	

c) F13: An incorrect request is made when an empty string is passed as a path variable. This causes the URL to not match any actual route, resulting in a 404 (Not Found) response for the `ts-basic-service` (F13a). For the `ts-travel-service` (F13b) and `ts-travel2-service` (F13c), the URL matches a route but with an incorrect HTTP method, leading to a 405 (Method Not Allowed) response.

d) F27: Certain endpoints take a UUID as a string from the URL path but do not confirm whether the provided argument is a valid UUID before conversion. If the input cannot be transformed to a UUID, the `UUID.fromString` method will throw an unhandled `IllegalArgumentException` [43]. This fault occurs in the `ts-assurance-service` (F27a), `ts-security-service` (F27b), `ts-auth-service` (F27c), `ts-food-service` (F27d), `ts-order-other-service` (F27e), and `ts-order-service` (F27f).

E. RQ3: Comparison of Faults with Current Research

1) Study Design: To label and compare the fault types we found in the TrainTicket system with current knowledge about possible faults in microservice systems, we first performed a small literature search to find fault taxonomies for microservice systems. Through this search, we found a general categorization of faults into symptoms and root causes proposed by Zhou et al. [13]. Additionally, we found a broad taxonomy of faults in microservice systems considering their influence on non-functional attributes like maintainability and security, published by Silva et al. [23]. Furthermore, we found a fault taxonomy focusing on integration-relevant faults in microservice systems, proposed by Gregor et al. [24] and taxonomy of issues in microservice systems by Waseem et al. [22]. We used those four classifications to label the faults we detected in TrainTicket to provide new examples for existing fault types or detect new fault types not represented in those classifications yet.

2) Study Procedure: To classify and compare our detected faults with fault groupings or taxonomies reported in related work, we first assess the methodology behind their classifications. Whenever descriptions in the research papers are too abstract to understand all of their detailed decisions, we take into account their replication packages. We try to find similar examples of faults where possible to validate our classifications. We additionally introduce confidence levels from c1-c3 to represent the degree of certainty in our classifications:

Whenever finding a suitable (bottom-level) category to classify a fault is impossible, we infer that we have found a new, non-reported fault type and include a new category in their taxonomy structure.

3) Study Results: The initial grouping of all faults into the root causes and influences proposed by Zhou et al. [13] is shown in Table IV. Most of the faults we found in the TrainTicket system fall into the category of functional internal faults. We only identified two faults to be functional interaction faults, one to be a non-functional internal fault and one to be a non-functional interaction fault. It is worth mentioning that none of the faults we identified were environment faults. However, this could be due to the fact that dependencies were updated and cleaned up by the students in the beginning of the project and before the test cases development. As presented in Table I, we could associate 8 faults directly with existing issue types or sub-categories in the taxonomy by Waseem et al. [22]. For the remaining 19 faults, we were able to find a suitable category and sub-category, but none of the concrete issue types fit the respective faults. Therefore, we infer that we found new fault types, not represented in the taxonomy yet and add new issue types to the taxonomy. This results in seven new issue types, namely *Logic Sequence Error*, *Data Overwrite Error*, *Condition Logic Error*, *Insufficient Value Validation*, *Initialization Error*, *Ignored Return Value*, and *Infinite Loop*.

The classification based on the taxonomy by Silva et al. [23] is presented in Table I as well. 20 faults could be classified directly into one of the original fault types. Six faults could be associated to a sub-category and characteristic but were not fully represented by the existing faults types there. Therefore, we again infer that we found new fault types, not represented in the taxonomy yet and add three new fault types for those faults: *Invalid Response Data Fault*, *Insufficient Credentials Accepted*, and *Faulty Authorization*. It is noteworthy that many faults were classified as *Internal Fault* (13/27), although they were classified into six different types of issues in Waseem et al.'s taxonomy [22]. This is due to the inherent nature of Silva et al.'s [23] taxonomy, as they put a focus on non-functional attributes and do not differentiate functional problems in detail.

The taxonomy by Gregor et al. [24] focuses solely on integration-relevant faults. Naturally, not all faults that we found within the TrainTicket system could be classified with this taxonomy, as they were internal faults. We could identify

TABLE IV
GROUPING OF THE FAULTS INTO THE ROOT CAUSES AND INFLUENCES PROPOSED BY ZHOU ET AL. [13]

Root Cause \ Influence	Functional	Non-Functional
Internal	F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F14, F17, F18, F19, F20, F21, F22, F23, F25, F27	
Interaction	F13, F24	F12, F15, F26, F16
Environment		

F12, F13, F15, F16, F24, and F26 as integration-relevant faults. Their classification is depicted in Table III. We were able to sort three of those faults directly into a subcategory within the taxonomy. Three faults could be related to a higher-level category but not to a sub-category fault type. Therefore, we added a new sub-category (*Unauthorized Access Granted*) to the taxonomy and infer that we found a new fault type not yet depicted in the taxonomy.

F. RQ4: Analysis of Commit History

1) Study Design: To assess whether the faults we detected in version `0.0.4` of TrainTicket were resolved in further development, we examined the commit history of the TrainTicket repository. This way, we are able to show possible fixes to those faults that were fixed and identify faults that are still prevalent in the current version of TrainTicket. Each fault was analyzed individually within the commit history, including those, that appeared in multiple services. Consequently, a fault could have been addressed within one service while the same fault type remained overlooked in another.

2) Study Procedure: To track how faults evolved in the code over time, we classify their final states into four categories:

- *Directly Resolved:* The fault was explicitly fixed in subsequent development if the code location of the fault was modified in a targeted manner, indicating that the developers were aware of the fault and intentionally addressed the issue.
- *Indirectly Resolved:* The faulty code was removed or refactored without clear evidence that the fault was intentionally addressed. This typically occured when a service or method was deleted, or broader design changes eliminated the issue as a side effect.
- *Partially Resolved:* If a fault type appeared multiple times in a service and only some instances were fixed, but at least one occurrence remains.
- *Unresolved:* The fault persists if the relevant code remains unchanged between version `0.0.4` and version `1.0.0`.

Identifying the status of each fault first involved directly comparing each fault location between version `0.0.4` and version `1.0.0`. As non-integration faults are confined to a specific line or section of code within a single microservice, their location can be clearly identified within that service. In contrast, integration faults pose a challenge in identifying their location, as no single, definitive service is responsible for the fault[44]. For example, if Service A sends an integer as a userId, but Service B expects a string, the fault could be resolved by either converting the integer in Service A before transmission or modifying Service B to accept an integer. Thus, for integration faults, we examined both sides of the interaction when analyzing source code changes between the two versions

TABLE V
CLASSIFICATION OF FAULTS BASED ON THEIR RESOLUTION STATUS

Status	Fault ID
Directly Resolved	F5a, F5b, F9a, F9b, F13a, F16a, F16b, F27b, F27c
Partially Resolved	F4a, F27e, F27f
Indirectly Resolved	F6, F11, F13b, F13c, F20a, F20b
Unresolved	F1a, F1b, F2, F3, F4b, F4c, F4d, F4e, F4f, F7, F8a, F8b, F8c, F10, F12, F14, F15a, F15b, F17a, F17b, F18a, F18b, F19, F21, F22, F23, F24, F25a, F25b, F25c, F26, F27a, F27d

3) Study Results: Our analysis of the commit history for the TrainTicket system revealed that out of the 51 identified faults, 9 were directly resolved, 3 were partially resolved, and 6 were indirectly resolved. The remaining 33 faults, however, remained unresolved in version `1.0.0` of the system. Table V presents the classification of each fault according to its resolution status. An excerpt of the faults is provided in the following to describe their resolution status in greater detail. We provide links to the specific commits that resolved each fault in the replication package [40].

a) F11 (Indirectly Resolved): The mismatch between two different data types (`HTTPEntitiy` vs. `ResponseEntity<Response>`) is no longer present in version `1.0.0`. However, the commit which resolved the fault deleted all occurrences of fallback methods. Therefore, whether this fault was recognized and intentionally addressed by its developers or resolved as a byproduct of broader changes remains unknown.

b) F13a (Directly Resolved):: F13a involved the `ts-basic-service` generating an empty string and passing it as a path variable to the `ts-price-service`, which resulted in an invalid request URL. This fault was resolved by modifying the implementation in the `ts-basic-service` to return a status 0 with an appropriate message if the value previously used as input for a request URL does not exist.

c) F13b and F13c (Indirectly Resolved):: Like F13a, F13b and F13c involved passing an empty string variable as a path to the `ts-ticketinfo-service`. This service was removed entirely, as well as the requests made to it, which indirectly resolved these faults.

d) F27b and F27c (Directly Resolved):: All faults related to F27 stemmed from failing to check whether a string could be converted to a UUID before attempting the conversion, leading to a `NumberFormatException`. In both the `ts-security-service` (F27b) and the `ts-auth-service` (F27c), these faults were directly addressed by changing the data type of identifiers from UUID to string, eliminating the need for conversion.

e) F27e and F27f (Partially Resolved): Similar to F27b and F27c, F27e and F27f were addressed by adjusting the data type of identifiers for the `Order` entities within both the `ts-order-other-service` and the `ts-order-service` from UUID to string. However, not all instances of unchecked `UUID.fromString()` operations were updated within the services, leaving these faults only partially resolved.

G. RQ5: Relation between Test Case Types and Detected Fault Types

1) Study Design: In RQ1, one component test suite and one integration test suite were created. Each of the test cases in the combined test suites furthermore falls into one of three categories: *Happy Path Test*, *Syntactic Test*, and *Edge Case Test*. Using the test levels and these three types of test cases, we classified each detected fault based on the type of test case that uncovered it.

2) Study Results: Figure 2 shows the distribution of faults that were detected by component or integration test cases. Out of the 51 discovered faults, 36 were detected solely by component tests, 10 solely by integration tests, and 5 by component and integration tests. Figure 3 provides a Venn diagram illustrating the relationship between test case categories and the faults they uncovered. 14 faults were uncovered solely by *Happy Path Tests*, while 29 were uncovered solely by *Edge Case Tests*. Additionally, five faults were detected by both *Happy Path Tests* and *Edge Case Tests*, and three faults were detected by all three types of test cases. Notably, no fault was uncovered solely by a Syntactic Test. The replication package [40] provides a detailed list of the exact test cases, their types, and the faults they revealed.

3) Interpretation of Results: Our results show that more faults, as well as a greater variety of fault types, were detected through the component test suite than through the integration test suite. This aligns with our previous findings (RQ3, RQ4), where most detected faults were classified as component faults, as they originate within individual services without or before affecting their interactions. Interestingly, some faults classified as integration-relevant (e.g., F12, F26) were only detected by component tests, suggesting that these faults already manifested at the component level before leading to observable integration issues. Conversely, faults F13, F14, F16, F20, and F24 were solely detected through integration tests, highlighting the necessity of testing service interactions explicitly. These findings confirm that both test suites contribute unique fault detection capabilities, reinforcing the importance of a combined testing approach in microservice systems. Our results furthermore indicate that our test suite is equally effective in uncovering faults without the *Syntactic Tests*. Their inability to uncover faults indicates that creating test cases without considering the system's semantics is insufficient for this specific microservice system. Effective fault detection requires an understanding of system behavior to design tests that account for its specific functionality. The fact that *Edge Case Tests* uncovered more than twice as many faults as *Happy Path Tests* suggests that the developers of the TrainTicket

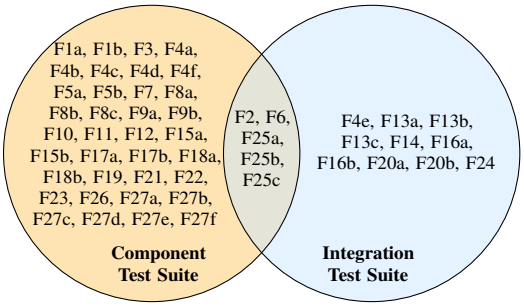

Fig. 2. Venn Diagram Showing Faults Uncovered by Each Test Level

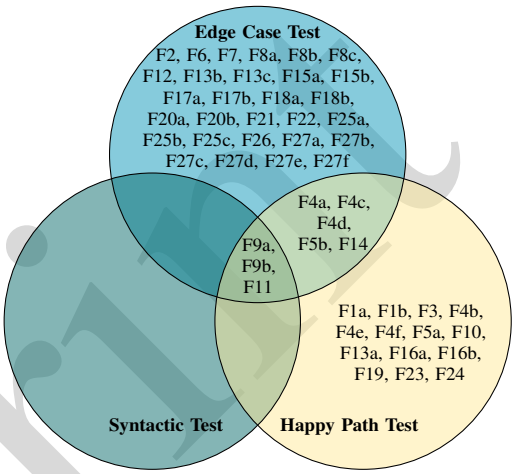

Fig. 3. Venn Diagram Showing Faults Uncovered by Each Type of Test Case

system may have prioritized ensuring the service's functionality with typical inputs over unusual ones. This seems reasonable, given that TrainTicket was developed as an example system where failures from unusual inputs would not impact real users. However, it is important to note that our differentiation between *Happy Path Tests* and *Edge Case Tests* should be viewed with caution. The definition of a "typical" input is subjective and can be stretched to some extent. Since TrainTicket lacks a formal specification, we had to infer expected behavior from implementation details and common best practices, which may not always be accurate. Moreover, without real users, there is no data on standard inputs, meaning some inputs classified as "typical" may actually be edge cases and vice versa.

IV. Related Work

Smith et al. [12] developed an E2E benchmark test suite for the TrainTicket system using Selenium for user simulation and Gatling for load testing. While their focus was on E2E testing, our benchmark targets component and integration testing. To capture diverse perspectives, we combined suites from four independent student teams, while Smith et al., developed their test suites themselves. Unlike Smith et al. [12], who also published a benchmark test suite for another open-source microservice system, our work focuses exclusively on the TrainTicket system to enable a detailed analysis of its faults, which was not the focus of their study.

Marculescu et al. [45] generated test suites with automated tools for eight REST API systems, differing from our focus on microservice architectures and manually written test suites. They also created a new fault taxonomy for REST APIs, whereas we classify faults using existing microservice taxonomies to enrich current datasets. Furthermore, we examine the link between the faults and the test cases that uncovered them, as well as the commit history of the system, to identify which faults were recognized and resolved by its developers. These topics were not explicitly addressed in their study.

Zhou et al. [13] introduced the TrainTicket system as a benchmark for microservice research. Their work involved a fault analysis through industrial surveys, in which 22 faults were identified and artificially replicated into their system. To classify their faults, Zhou et al. [13] proposed a grouping into root causes and influences. However, fault analysis and classification were not the primary focus of their study – instead, they focused on evaluating debugging effectiveness. In contrast, our work emphasizes the classification of faults using existing taxonomies to add new insights to them. Additionally, we analyze naturally occurring faults uncovered in the TrainTicket system instead of artificially created faults derived from industrial surveys.

In contrast to the fault or issue taxonomies [22]–[24] that we used to classify our detected faults, we did not propose a new taxonomy but added onto those existing taxonomies. Furthermore, we did not analyze already reported issues from git repositories [22], literature [23] or interviews with practitioners [24] but created test suited and identified faults in an open-source system. Therefore, we were also able to analyze the relation between types of test cases and faults detected, which they were not.

V. Threats to Validity

A. Internal Validity

A potential threat to the internal validity of the benchmark test suites is that they were initially developed by students. Nevertheless, their work holds significant value. The students' lack of prior familiarity with the system allowed for unbiased testing, leading to the discovery of 51 faults and 8 atypical design decisions overlooked by the original developers. Their adherence to structured test selection principles ensured methodological rigor, and the involvement of multiple teams introduced diverse perspectives, increasing fault detection. Student involvement also supports reproducibility and scalability for future research. Finally, the systematic consolidation of test cases ensures that the most effective and relevant tests are retained, while redundant test cases are excluded, improving the overall quality of the resulting test suites. A further threat to validity stems from the lack of a formal (business) specification for TrainTicket. Consequently, some faults were identified based on common design principles or inferred requirements, introducing a degree of subjectivity. Nonetheless, many faults were clearly evident from context, even if they only caused system errors rather than failures. For instance, in the ts-auth-service, account creation is allowed with empty or overly short passwords because the length check occurs after encoding. Since encoded passwords always exceed the minimum length, invalid inputs are not rejected. As this check was explicitly implemented, we assumed it was misplaced rather than obsolete – though in either case, it constitutes a fault or code smell, as redundant or incorrectly placed code should usually be avoided.

B. External Validity

Since TrainTicket was developed as a benchmark system for research [13] without actual users or business value, its faults may not fully represent those found in industry-scale microservice systems. However, the goal of this study was not to compile a comprehensive catalog of faults in microservice systems but rather to identify the types of faults present in TrainTicket. This information can support future research using the system for evaluation purposes. Our fault detection results were used to expand existing knowledge on microservice faults, but we do not claim that these findings are generalizable to all microservice systems. Differences in service functionality, complexity, and operational environments may lead to distinct fault patterns, requiring different testing approaches. Consequently, the generalizability of our insights into the relationship between test types and detected fault types remains uncertain. Nevertheless, TrainTicket is a well-established system that has been widely used in previous studies [6], [13]–[21], which makes it a reasonable choice for this work.

VI. Conclusion

We presented benchmark test suites on the component and integration levels, developed through a case study on the widely-adopted TrainTicket microservice system [13]. To ensure diverse perspectives, four independent teams created initial test suites, which we subsequently consolidated into two unified and extensive test suites—one per test level. Using these test suites, we identified 51 faults in TrainTicket and analyzed them in the context of existing fault taxonomies for microservices. This comparison revealed 11 gaps in three existing taxonomies for which we propose new fault categories. Further analysis showed that 36 of the 51 detected faults were detected solely by the component test suite, while 10 were detected solely by the integration test suite. Our findings confirm that both test suites contribute unique fault detection capabilities, highlighting the importance of a multi-level testing approach in microservice systems and offering practical guidance for real-world testing strategies aiming to uncover a broader range of faults. Developers of microservice systems – especially those lacking formal specifications – can use the benchmark as a reference for test creation. Our openly available test suites and fault data [40] provide a foundation for evaluating automated test generation and assessment techniques, serve as a reusable benchmark for future studies, and offer a practical resource for education and training in microservice testing. Finally, by identifying new fault types and extending existing taxonomies, our study contributes to a deeper understanding of faults in microservice systems and offers valuable resources for both researchers and practitioners.

References

[1] J. Lewis and M. Fowler, *Microservices*, https://martinfowler.com/articles/microservices.html, Mar. 2014. (visited on 08/25/2022).

[2] S. Newman, *What are microservices?* 1st edition. O'Reilly Media, Inc., 2016.

[3] Software AG, *Do you utilize microservices within your organization?* In Statista. https://www.statista.com/statistics/1236823/microservices-usage-per-organization-size/, Apr. 2021. (visited on 09/19/2024).

[4] IBM, *Applications using microservices worldwide in 2021*, In Statista. https://www.statista.com/statistics/1236542/applications-using-microservices-list/, Apr. 2021. (visited on 09/19/2024).

[5] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou, and Z. Li, "Microservices: Architecture, container, and challenges," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2020, pp. 629–635. DOI: 10.1109/QRS-C51114.2020.00107.

[6] L. Giamattei, A. Guerriero, R. Pietrantuono, and S. Russo, "Automated Grey-Box Testing of Microservice Architectures," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2022, pp. 640–650. DOI: 10.1109/QRS57517.2022.00070.

[7] A. Arcuri, "RESTful API Automated Test Case Generation," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Jul. 2017, pp. 9–20. DOI: 10.1109/QRS.2017.11.

[8] A. Arcuri, "EvoMaster: Evolutionary Multi-context Automated System Test Generation," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2018, pp. 394–397. DOI: 10.1109/ICST.2018.00046.

[9] A. Arcuri, "RESTful API Automated Test Case Generation with EvoMaster," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 1, 3:1–3:37, Jan. 2019. DOI: 10.1145/3293455.

[10] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API Fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 748–758. DOI: 10.1109/ICSE.2019.00083.

[11] E. Viglianisi, M. Dallago, and M. Ceccato, "RESTTESTGEN: Automated Black-Box Testing of RESTful APIs," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, Oct. 2020, pp. 142–152. DOI: 10.1109/ICST46399.2020.00024.

[12] S. Smith, E. Robinson, T. Frederiksen, *et al.*, "Benchmarks for End-to-End Microservices Testing," in *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, Jul. 2023, pp. 60–66. DOI: 10.1109/SOSE58276.2023.00013.

[13] X. Zhou, X. Peng, T. Xie, *et al.*, "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, Feb. 2021. DOI: 10.1109/TSE.2018.2887384.

[14] P. Liu, H. Xu, Q. Ouyang, *et al.*, "Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2020, pp. 48–58. DOI: 10.1109/ISSRE5003.2020.00014.

[15] V. Cortellessa, D. Di Pompeo, R. Eramo, and M. Tucci, "A model-driven approach for continuous performance engineering in microservice-based systems," *Journal of Systems and Software*, vol. 183, p. 111 084, Jan. 2022. DOI: 10.1016/j.jss.2021.111084.

[16] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "Causal Inference Techniques for Microservice Performance Diagnosis: Evaluation and Guiding Recommendations," in *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, Sep. 2021, pp. 21–30. DOI: 10.1109/ACSOS52086.2021.00029.

[17] X. Zhou, X. Peng, T. Xie, *et al.*, "Delta debugging microservice systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, New York, NY, USA: Association for Computing Machinery, Sep. 2018, pp. 802–807. DOI: 10.1145/3238147.3240730.

[18] X. Zhou, X. Peng, T. Xie, *et al.*, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 683–694. DOI: 10.1145/3338906.3338961.

[19] A. Walker, I. Laird, and T. Cerny, "On Automatic Software Architecture Reconstruction of Microservice Applications," in *Information Science and Applications*, Singapore: Springer, 2021, pp. 223–234. DOI: 10.1007/978-981-33-6385-4_21.

[20] Z. Li, J. Chen, R. Jiao, *et al.*, "Practical Root Cause Localization for Microservice Systems via Trace Analysis," in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, Jun. 2021, pp. 1–10. DOI: 10.1109/IWQOS52092.2021.9521340.

[21] X. Hou, J. Liu, C. Li, and M. Guo, "Unleashing the Scalability Potential of Power-Constrained Data Center in the Microservice Era," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP '19, New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 1–10. DOI: 10.1145/3337821.3337857.

[22] M. Waseem, P. Liang, M. Shahin, A. Ahmad, and A. R. Nassab, "On the Nature of Issues in Five Open Source Microservices Systems: An Empirical Study," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '21, New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 201–210. DOI: 10.1145/3463274.3463337.

[23] F. Silva, V. Lelli, I. Santos, and R. Andrade, "Towards a Fault Taxonomy for Microservices-Based Applications," in *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, ser. SBES '22, New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 247–256. DOI: 10.1145/3555228.3555245.

[24] L. Gregor, A. Hentschel, L. Kastner, and A. Pretschner, "A Taxonomy of Integration-Relevant Faults for Microservice Testing," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2025, pp. 138–149. DOI: 10.1109/ICST62969.2025.10989000.

[25] A. S. Abdelfattah, T. Cerny, J. Y. Salazar, *et al.*, "End-to-End Test Coverage Metrics in Microservice Systems: An Automated Approach," in *Service-Oriented and Cloud Computing*, ser. Lecture Notes in Computer Science, Cham: Springer Nature Switzerland, 2023, pp. 35–51. DOI: 10.1007/978-3-031-46235-1_3.

[26] I. Ghani, W. M. N. Wan-Kadir, A. Mustafa, and M. I. Babir, "Microservice Testing Approaches: A Systematic Literature Review," *International Journal of Integrated Engineering*, vol. 11, no. 8, pp. 65–80, Dec. 2019. [Online]. Available: https://publisher.uthm.edu.my/ojs/index.php/ijie/article/view/3856.

[27] "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary," *ISO/IEC/IEEE 24765:2017(E)*, pp. 1–541, Aug. 2017. DOI: 10.1109/IEEESTD.2017.8016712.

[28] F. Compagno and S. Borgo, "Ontological Analysis of Malfunctions: Some Formal Considerations," in *Formal Ontology in Information Systems*, IOS Press, 2024, pp. 149–162. DOI: 10.3233/FAIA241302.

[29] A. Avizienis, J.-C. Laprie, and B. Randell, "Fundamental Concepts of Dependability,"

[30] A. Pretschner, *Defect-based Testing* (NATO science for peace and security series - d: information and communication security v. 50). IOS Press, 2017.

[31] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, Apr. 2009. DOI: 10.1007/s10664-008-9102-8.

[32] G. Bath and J. McKay, *The Software Test Engineer's Handbook: A Study Guide for the ISTQB Test Analyst and Technical Test Analyst Advanced Level Certificates 2012*. Rocky Nook, Inc., Jun. 2014.

[33] L. Gupta, *HTTP Methods*, May 2018. [Online]. Available: https://restfulapi.net/http-methods/ (visited on 04/10/2025).

[34] L. Gupta, *HTTP Status Codes*, May 2018. [Online]. Available: https://restfulapi.net/http-status-codes/ (visited on 04/10/2025).

[35] S. Bruning, S. Weissleder, and M. Malek, "A Fault Taxonomy for Service-Oriented Architecture," in *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, Plano, TX, USA: IEEE, Nov. 2007, pp. 367–368. DOI: 10.1109/HASE.2007.46.

[36] B. Beizer, *Software testing techniques*. Van Nostrand Reinhold, 1990.

[37] H. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Proceedings. Conference on Software Maintenance 1990*, Nov. 1990, pp. 290–301. DOI: 10.1109/ICSM.1990.131377.

[38] C. Wei, L. Xiao, T. Yu, *et al.*, "Automatically Tagging the "AAA" Pattern in Unit Test Cases Using Machine Learning Models," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, Rochester MI USA: ACM, Oct. 2022, pp. 1–3. DOI: 10.1145/3551349.3559510.

[39] jpreese, *Best practices for writing unit tests - .NET*, Nov. 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices (visited on 03/20/2025).

[40] *Repository with Benchmark Test Suites and All Replication Package Material*. 2025. DOI: 10.5281/zenodo.15646869.

[41] S. H. Edwards and Z. Shams, "Do student programmers all tend to write the same software tests?" In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, ser. ITiCSE '14, New York, NY, USA: Association for Computing Machinery, Jun. 2014, pp. 171–176. DOI: 10.1145/2591708.2591757.

[42] A. Hora, "Test Polarity: Detecting Positive and Negative Tests," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024, New York, NY, USA: Association for Computing Machinery, Jul. 2024, pp. 537–541. DOI: 10.1145/3663529.3663793.

[43] *UUID (Java Platform SE 8)*. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html (visited on 03/20/2025).

[44] A. Pretschner and L. Gregor, "Understanding Integration Testing," in *Engineering Safe and Trustworthy Cyber Physical Systems–Essays Dedicated to Werner Damm on the Occasion of His 71st Birthday*, 2024. [Online]. Available: https://mediatum.ub.tum.de/doc/1726159.

[45] B. Marculescu, M. Zhang, and A. Arcuri, "On the Faults Found in REST APIs by Automated Test Generation," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 3, 41:1–41:43, Mar. 2022. DOI: 10.1145/3491038. (visited on 10/25/2022).