A Taxonomy of Integration-Relevant Faults for Microservice Testing

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

Anja Hentschel
Siemens AG
Munich, Germany
anja.hentschel@siemens.com

Leon Kastner
School of CIT
Technical University of Munich
Munich, Germany
leon.kastner@tum.de

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

***Abstract*—Microservices have emerged as a popular architectural paradigm, offering a flexible and scalable approach to software development. However, their distributed nature and diverse technology stacks introduce inherent complexities, surpassing those of monolithic systems. The integration of microservices presents numerous challenges, from communication failures to compatibility issues, compromising system reliability. Understanding faults in these distributed components is crucial for preventing defects, devising test strategies, and implementing robustness testing. Despite the significance of these software systems, existing taxonomies are limited, as they primarily focus on non-functional attributes or lack empirical validation. To address these gaps, this paper proposes an extensive taxonomy of the most common integration-relevant faults observed in large-scale microservice systems in industry. Leveraging insights from a systematic literature review and ten semi-structured interviews with industry experts, we identify common integration-related faults encountered in real-world microservice projects. Our final taxonomy was validated through a survey with an additional set of 16 practitioners, confirming that almost all fault categories (21/23) were experienced by at least 50% of the survey participants.**

***Index Terms*—microservice systems, fault taxonomy, service integration, software testing, real faults**

I. Introduction

The microservice architectural style is a methodology for developing an application as a collection of small services. Each service runs in its own process and interacts with others through lightweight mechanisms, such as REST APIs or message queues. These services are built around specific business functions and can be deployed independently, often using automated deployment methods. Management and development of the services is decentralized, allowing for flexibility in choosing programming languages and data storage technologies for each service. [1], [2]

Due to its advantages, microservice architecture has gained widespread adoption in practice [3], [4]. However, the distributed nature of microservices, coupled with their diverse technology stack, introduce inherent complexities for building robust systems, often surpassing those encountered in monolithic systems [5]. Integrating microservices with each other introduces various potential pitfalls, ranging from communication failures to compatibility issues, which can compromise the reliability and robustness of the system as a whole. Knowledge about potential integration faults is thereby crucial for ensuring the quality, security, and correct functioning of such systems. Accordingly, this information would allow creating integration-fault-aware test strategies or test cases (e.g., through defect-based testing [6]), implement robustness testing through fault injection, or create mutation operators to evaluate the quality of existing test suites.

For other system types, such as cyber-physical systems or object-oriented monoliths, research has already acknowledged and acted on the need for integration fault taxonomies [7]–[10]. However, for microservice systems, knowledge about these faults is still sparse. Integration fault taxonomies from those other contexts are missing important aspects that come with the distributed nature of microservice systems, e.g., having components like message queues that realize the communication between services and can have constraints or malfunction. Existing taxonomies pertaining to service-oriented architecture (SOA) or web services [11], [12] rely on theoretical frameworks, lacking empirical data and validation.

To the best of our knowledge, only one taxonomy has been proposed in the literature for microservice-based systems by Silva et al. [13]. It includes a mixture of integration faults and faults that only affect the functionality of single services. The taxonomy also primarily examines the impact of faults on non-functional attributes such as maintainability. However, testing predominantly focuses on functional aspects.

Additionally, knowledge about the severity of faults and the effort that it takes to fix them can help with prioritization during test strategy creation or testing. However, current fault taxonomies do not provide this information.

To address these limitations and advance our understanding of integration faults in microservice systems, this research paper proposes the following approach: We conducted a systematic literature review to leverage existing knowledge related to integration faults in service-based systems. Then, we conducted ten semi-structured interviews involving industry experts and practitioners with knowledge from over 20 different microservice projects from various domains to deepen our understanding. Based on those two steps, we created a taxonomy with integration-relevant faults encountered in real-world microservice systems. We additionally validated the resulting taxonomy through a survey with a bigger set of practitioners and experts. We also collected information on the

979-8-3315-0814-2/25 © 2025 IEEE

ICST 2025, Naples, Italy
Technical-Research Track

perceived severity and the effort to fix those faults.

Our main contribution to the state of the art in microservice testing is the validated taxonomy of integration-relevant faults for microservice systems commonly observed in practice. To the best of our knowledge, this is the first taxonomy that focuses on integration faults in microservice systems and includes interviews with practitioners. The work presented in this paper enables several important directions for future work, such as:

Systematic Fault Identification. Fault taxonomies categorize different types of software faults, making it easier to systematically identify and categorize faults in software systems. The works of Islam et al. [14] and Zheng et al. [15] have shown how other fault taxonomies or classification schemes were used to classify and analyze bugs and faults in software systems, allowing insights into fault patterns and evaluating fault detection approaches.

Creating Labeled Datasets. In addition to classifying and categorizing software faults, a structured taxonomy can be leveraged to create labeled datasets that are essential for machine learning (ML) approaches. By organizing faults into distinct categories, projects with similar fault patterns can be mined to generate comprehensive datasets. These labeled datasets can then be used to train ML models for bug prediction and other tasks.

Test Case Design. Fault taxonomies can guide the creation of test cases using the defect-based approach[6], ensuring that tests are designed to cover a wide variety of fault types. By aligning test cases with known categories of faults, testers can better anticipate and detect issues in the software.

Risk Assessment. The information on perceived severity and effort to fix certain fault categories allows testers to prioritize testing efforts on fault categories with more problematic implications. This can also help to prioritize efforts to mitigate the risk of certain fault categories already at design time.

Enhancing Software Reliability. Our taxonomy can furthermore be used for fault injection approaches [16]–[18] to test the robustness and reliability of a microservice system.

Enhancing Manual Repair Processes. In the absence of automated techniques, understanding the root causes of bugs can significantly help developers in manually fixing programs. By increasing their awareness of bug origins, developers can better prioritize which potential causes to inspect, using the relative importance of these causes as outlined in our taxonomy.

Mutation Testing. Taxonomies with common faults can be used to create mutation operators for mutation testing approaches to evaluate the quality of test cases [19].

II. Methodology

To create an extensive fault taxonomy including realistic integration-relevant faults, we first started by analyzing the existing knowledge from the literature. Through a systematic literature review, which is reported in Sec. II-A, we found a fault taxonomy for SOA which we used as a basis for our taxonomy. We then adapted the existing top-level fault categories of this taxonomy to the microservice context (Sec. II-B). To enrich and validate the taxonomy and also deepen our understanding, we conducted semi-structured interviews with practitioners (Sec. II-C). Lastly, we validated the resulting taxonomy through a survey with another set of practitioners (Sec. II-D). Through this survey, we also collected data on the perceived severity of the faults and the effort to fix them. The artifacts created in our studies are provided in a replication package [20] for reference.

A. Systematic Literature Review

To find our initial taxonomy, we performed a systematic literature review using three different search engines: Scopus, IEEE Explore, and ACM Library. As a search for fault taxonomies for microservice systems only resulted in one publication, we decided to broaden our search to SOA systems as well. The search string we used for all three search engines is as follows: *(“fault taxonomy” OR “fault ontology”) AND (microservice* OR “micro-service*” OR “micro*service” OR soa* OR “service-oriented*” OR “service*oriented*”)*. For Scopus, we restricted the search to titles, keywords, and abstracts, and for the document types “article” and “conference paper”. For the ACM library, we restricted the document type to “research article”. This resulted in 10 papers in Scopus, 1 in IEEE Explore, and 8 in ACM Library. References to all publications found through the literature research and details on the further selection are reported in the replication package [20]. After removing duplications, our search resulted in a total of 16 papers. Based on the abstracts of these papers, we applied the following inclusion and exclusion criteria:

- **Inclusion:** Studies are considered if they propose a taxonomy or ontology of faults in SOA or microservice systems.
- **Exclusion:** Studies were excluded from further review if they were focused on other system types or did not propose a fault taxonomy.

This resulted in a total of 5 publications. Next, we applied the same criteria to the full text of each of the papers. This resulted in four publications: three including taxonomies for SOA systems [11], [12], [21] and one including a taxonomy for microservice systems [13]. Silva et al. [13] propose the only existing fault taxonomy for microservice-based systems, focusing on how faults influence non-functional attributes like maintainability and performance. While this taxonomy provides valuable insights, its variability in structure, granularity, and generalizability poses challenges for directly using it to develop fault injection techniques or test strategies. Additionally, the taxonomy is not centered around integration-relevant faults and, therefore, is not usable for evaluating integration or system-level tests.

From the SOA-related publications, we chose the taxonomy of Bruning et al. [11] as it is easy to understand, well described, and provides a structured overview with multiple fault category levels with mostly integration-relevant faults. Additionally, according to Scopus, it has the highest number of citations in our pool of papers. The taxonomy of Bruning et al. [11] was published in 2007 and, therefore, might be outdated, which may raise concerns about its alignment with the present-day

landscape of SOA. Since our work does not aim to explore the differences between SOA and microservice systems but rather uses this taxonomy as a basis in the absence of comparable taxonomies for microservice-based systems, we believe this is not a significant limitation.

B. Original Taxonomy and Adaptations

As the original taxonomy by Bruning et al. [11] was created for SOA services and, therefore, also uses terminology specific to the SOA context, we need to perform minor adaptations to the top-level categories, to fit the microservice context[1].

The original taxonomy [11] is built around the five steps that a SOA service needs to undertake every time it is executed: Publishing, Discovery, Composition, Binding, and Execution. Each of these phases is used as a top-level fault category and is subdivided into two to four more detailed sub-fault categories in the original taxonomy. To derive our adapted taxonomy, we analyzed each of the top-level fault categories' relevance for the microservice context based on knowledge from the literature. The parts of the original taxonomy that are discussed in the following, are shown in Fig. 1. All adaptations to the two first layers of the taxonomy are shown in Fig. 2.

Publishing Fault. In their work, Bruning et al. [11] explain that during the publishing phase, a service is deployed to a server, and its description is made publicly accessible. They emphasize that services must be self-descriptive, meaning that the description should encompass all necessary syntactic and semantic details about the service.

In the context of microservices, the term "publishing" is often used interchangeably with "deploying", which does not align with how Bruning et al. define "publishing". Additionally, the sub-category *Service Deployment Fault* under the broader *Publishing Fault* category implies that the authors distinguished between service publishing and service deployment within the SOA context. Although the *Publishing Fault* category does not perfectly suit the microservice environment, the related faults *Service Description Fault* and *Service Deployment Fault* and their associated sub-categories remain relevant since we also deal with service descriptions and deployments in this context. Therefore, we remove the top-level *Publishing Fault* category and elevate *Service Description Fault* and *Service Deployment Fault*, along with their respective sub-categories, to top-level faults in the adapted taxonomy.

Discovery Fault. In the SOA discovery phase, a consumer dynamically searches for a service that offers the required functionality [11]. To the best of our knowledge, this has not yet been achieved in real-world microservice applications. Instead, developers typically select the service they want to use, whether internal or third-party. In the case of microservices, static addresses are uncommon, particularly when multiple instances of the same service are deployed, and the communication between them must be dynamically defined [23]. There are multiple service discovery patterns mentioned in the literature and used in practice (e.g., client-side or server-side discovery [23]) that include a service registry component that helps identify the correct IP address of a service or service instance [23]. In such an architecture pattern, the faults *No Service Found* and *Wrong Service Found* seem relevant, which will be tested through the interviews reported in Sec. II-C. Therefore, we keep these fault classes for now, although the Discovery phase in a microservice context is slightly different than in a SOA context.

Composition Fault. In their fault taxonomy, Bruning et al. assume that composition in a SOA system operates dynamically, despite acknowledging in their paper that this has not yet been implemented in practice [11]. To the best of our knowledge, the dynamic composition of multiple microservices to deliver a combined functionality is still not feasible in real-world scenarios. However, since microservices typically rely on other microservices to provide a combined functionality [2], we believe that composition faults remain relevant in the context of microservices.

Binding Fault. In SOA, the binding phase occurs when the consumer binds itself to the service provider to initiate execution [11]. In the context of microservices, the term "binding" does not quite fit when using a REST API as a communication mechanism between two or more services. As a result, we rename this phase to "Connection", leading to the fault category *Connection Fault* with its sub-categories *Connection Denied* and *Connection to Wrong Service*.

Execution Fault. In the execution phase, the consumer sends input parameters to the provider, which processes the request and returns the output parameters to the consumer [11]. This phase is directly applicable to the microservice context without modification.

C. Interviews with Practitioners

To get deeper insights into integration-relevant faults in microservice systems in practice and enrich and further adapt the original taxonomy for the microservice context, we conducted semi-structured interviews with 10 practitioners.

1. Participant Recruitment. We considered practitioners with at least three years of working experience in projects with microservice systems as the target group of our interview study. To make our results as generalizable as possible, we opted to have as many participants from different companies and projects as possible. First, we contacted interview candidates who fulfilled these criteria from the author's personal contacts. Second, we asked them to disseminate the study to interested and qualified colleagues. In the end, this resulted in 10 successfully conducted interviews. The interviewees had between 4-32 years of work experience in software development (median = 17.5), with 3-8 years of experience in microservice projects (median = 5). Each of them had worked in 2 to 10 different microservice projects at the time of our interview. Their roles within these projects included test manager, test architect, test consultant, code quality consultant, DevOps consultant, DevOps engineer, DevOps architect, senior developer, and project lead.

[1]Parts of our preliminary adaptations were presented at the Microservices 2023 conference [22], a venue open for discussion of early-stage works with experts in the field, helping to shape the direction of potential later publications. We used this presentation to gather initial feedback on our efforts to develop the taxonomy which, therefore, should not be confused with a formal publication.

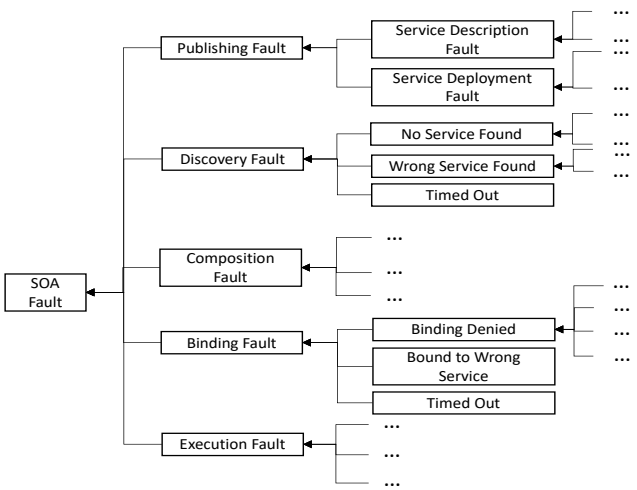

Fig. 1. Relevant section from the original taxonomy for SOA faults [11]

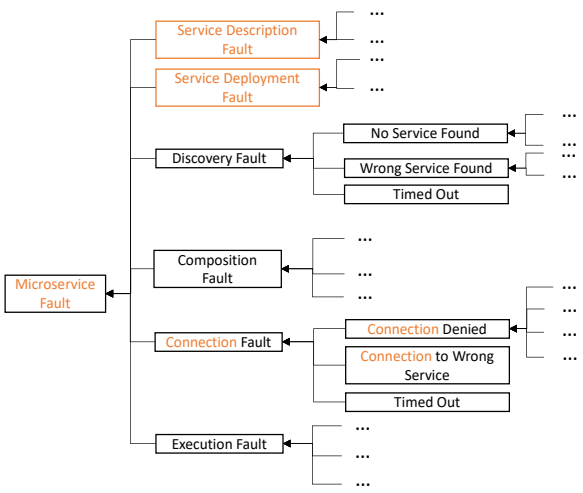

Fig. 2. Relevant section from the adapted taxonomy for the microservice context

The microservice projects they acquired knowledge from also came from various domains, including energy distribution, health care, the Internet of Things, industry automation, public administration, retail, insurance, and finance.

2. Interview Process. Although we are not creating a taxonomy from scratch, but on the basis of an existing taxonomy for SOA systems [11], we still want the interview questions to be as generic as possible. By doing this, we plan to get unbiased data that helps to not only enrich but also validate the adapted taxonomy. We opt for a semi-structured interview and use the questionnaire of Holling et al. [24] as a guideline for creating our interview questions as they have shown their approach to be effective for the elicitation of defects [24]. Semi-structured interviews demand the interviewer to formulate new questions based on the interviewee's responses, which might be challenging. Therefore, having an additional interviewer available to pose follow-up questions and support the primary interviewer when necessary can be beneficial [25]. As a result, our interviews are conducted by two authors simultaneously, each assigned distinct roles: one author led the interview, while the other author asked supplementary questions only when relevant. Our interviews focus on getting information about faults that can happen during the integration of multiple services within a microservice system. To ensure that all interviewees know the focus of our interviews and have the same understanding of integration as we have, an invitation with a description of the purpose and context of the interviews was sent to each of the interviewees beforehand. Additionally, we send them the three main questions of our interviews with this invitation to give them the chance to prepare for the interviews to get as much information from them as possible. All interviews are recorded and later transcribed for the data analysis.

3. Data Analysis. As it is commonly done in qualitative data analysis [26], we use a hybrid approach that combines deductive and inductive coding to analyze the data from our interviews. Utilizing the fault categories from the adapted taxonomy as the pre-defined set of codes or *codebook*, we first perform deductive coding [27]. This is an analysis method where qualitative data is labeled with existing codes. For text blocks that report faults not represented by any of the pre-defined codes, we perform inductive coding [28], [29] to create new fault categories for our fault taxonomy. The two interviewers were assigned as evaluators for each interview. Each evaluator proceeds with the coding process individually. Then, after the deductive and inductive coding of all interviews is completed, a meeting with both evaluators is conducted where differences between the coding are detected and resolved through discussions.

D. Taxonomy Construction

Each label created during the inductive coding phase is sorted into the structure of the adapted taxonomy, ensuring that the categories and sub-categories follow a "is a" relationship. After evaluating all interviews, we delete fault categories from the original taxonomy that are not represented in our interviews. Wherever a fault category only has one sub-category left after the construction and deletion process, it is deleted, and the sub-category is set in its place.

E. Taxonomy Validation

To ensure that the final taxonomy is comprehensive and representative of real faults in microservice integration, we conducted a validation process utilizing a survey with a distinct group of practitioners, separate from those engaged in the initial interviews.

The target group of this survey were practitioners with a minimum overall software development experience of three years and a minimum experience in microservice projects of one year. First, we invited selected practitioners who fulfilled these criteria from our personal contacts. Second, we asked them to

disseminate the study to interested and qualified colleagues. Third, we posted the invocation for the survey in a company internal forum to attract qualified participants from as many different projects and system domains as possible.

In total, 16 participants took part in our survey with a minimal overall software development experience of 6 years and a maximum of 35 years (median = 17.5). Regarding experience in microservice systems, the minimum was 2 years and the maximum was 22 years (median = 7). The participants have worked in microservice projects from various different domains, including smart infrastructure, IOT, finance, healthcare, energy distribution, tax and accounting, e-commerce, and automotive. The job descriptions they reported included software architect, software development, test architect, product owner, tester, software development and operations, architecture consulting, and consulting on continuous deployment.

We used Microsoft Forms[2] to create our survey form. We started the survey with the same background questions as in our interviews. Then, we proceeded with questions regarding our final taxonomy. Putting the whole taxonomy with all its leaves into a single survey page would make the survey hard to read and comprehend. Therefore, we partitioned it at the highest categories (e.g., "Service Description Faults"). In order to keep the time required to complete the survey to a reasonable level, we did not ask about all possible subcategories individually. Instead, we asked for the superordinate category for broad subcategories and only mentioned the individual leaf faults as examples. For example, for the "Service/ Description Mismatch" category, we added the question, "Did you ever encounter a mismatch between a service and its description?" and then added its descendant categories as examples of this fault in the description text. Additionally to the questions whether or not the participants have ever encountered the respective fault category in a project, we also added two more questions: (1) If the problem occurred, how severe was it, and (2) if it occurred, how much effort was required to fix it? With these additional two questions, we want to investigate the perceived severity and effort of the individual fault categories. This information can help prioritize one category over another when, i.e., creating a test strategy to efficiently direct resources. In the final part of the survey, we added a free-text answer, where participants could mention any other problems or faults related to the integration of multiple microservices that they did not find in the survey already. By doing this, we check whether our taxonomy already includes all faults encountered by developers in practice. If not, we find out what is still missing.

III. Results

In the following, we describe and explain the final taxonomy (displayed in Fig 3) as well as the results from our validation study.

A. Final Taxonomy

1. Service Description Faults. This fault category includes faults where either the description of a service (e.g., the OpenAPI specification) is incorrect or it does not match the related service. Both categories were divided into multiple sub-categories:

1.1 Description Incorrect. An incorrect description can be due to a *Format Fault* or a *Content Fault*. Faults caused by an incorrect description can be detected by checking the description of a service only. An example of a *Format Fault* would be a syntactic fault in the JSON format in an OpenAPI specification of a microservice. In the interviews, the participants mentioned unclear or ambiguous descriptions as a *Content Fault*. Those then led to misinterpretations and incompatible interfaces of services.

1.2 Service/Description Mismatch. This fault category considers faults where a service and its description do not match. For the *Description Incomplete* fault, the service includes some functionality or aspects that are not documented in the description. The interviewees mentioned missing information about the units and interpretation of response data here. This causes issues in understanding all service functionalities and can lead to misinterpretation and incompatible interfaces. The description of a feature (e.g., the functionality of an endpoint or the interpretation of a parameter or return type) can be wrong (*Wrong Feature Description*), or a feature defined by a service description can be incorrectly implemented (*Wrong Feature Implementation*). When the client of a service misinterprets the features of the used service, it is called a *Wrong Feature Interpretation*. In the interviews, there were examples of corner cases (e.g., empty lists) that one service expected the other to handle suitably, which was not the case. Some interview participants also mentioned that shared functionality of services was outsourced to shared libraries to reduce duplicated code. This *Implicit Interface* became problematic later when it was unclear how many and which services were affected by changes in these libraries.

2. Service Deployment Faults. These faults occur when a service is not successfully deployed.

2.1 Required Resource Missing. First, the deployment of a service can fail or be incorrect due to required resources being missing. Examples of such resources from the interviews were databases or configuration services. The service might be deployed but will fail to perform.

2.2 Wrong Configuration. The configuration of a microservice can be incorrect, so the deployment is not successful. The interviewees described various different examples here, e.g., ports or access rights that are not set or incorrectly set.

2.3 Service/Environment Incompatible. The service and its environment (e.g., the database, libraries, connection components like a message queue, containerization, deployment environment) are incompatible. The interviewees mentioned examples where service and environment were incompatible because they were not configured correctly from the start. Additionally, they described situations where a third party (e.g., AWS) changed or updated the environment, making the service and environment incompatixble, although it was compatible before.

[2]https://forms.office.com

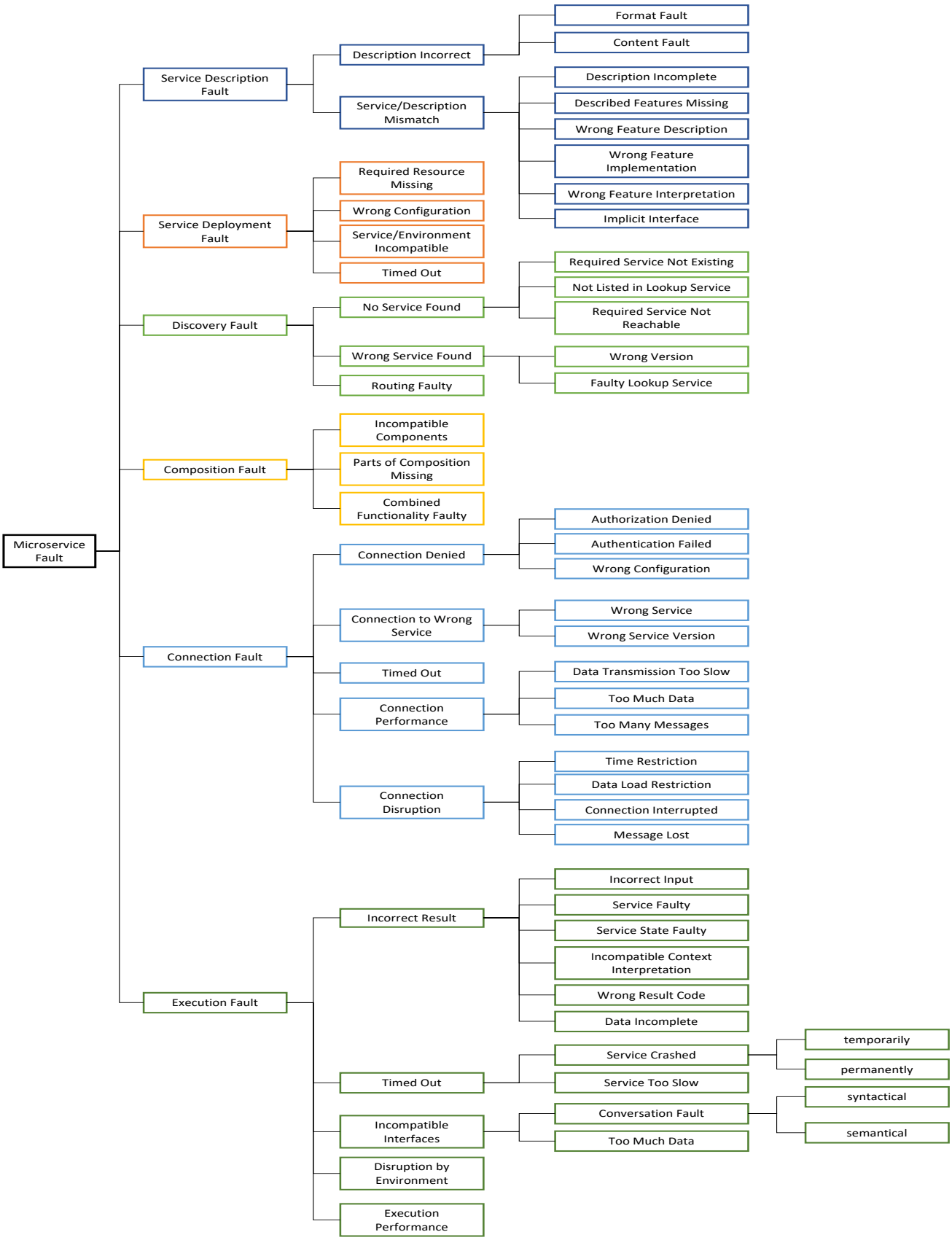

Fig. 3. The final taxonomy

2.4 Timed Out. Deployment of a service can run into a time out.

3. Discovery Faults. Faults during the discovery of another service (e.g., through a service registry component) or in the resulting service.

3.1 No Service Found. A required service cannot be found either because the *Required Service* is *Not Existing* (e.g., because it was not deployed or the deployment failed) or because it was *Not* (correctly) *Listed in* the *Lookup Service*.

3.2 Wrong Service Found. Not only no service can be found, but also the wrong service. This fault can occur either due to a *Faulty Lookup Service* which returns an entirely wrong service, or the *Wrong Version* of a service can be returned. The wrong (and therefore incompatible) version of another service can be found, either because both service versions (e.g., for different client/ consumer versions) are deployed in the system or only the wrong version of the service is deployed.

3.3 Routing Faulty. A (error) message is not correctly routed through a microservice system.

4. Composition Faults. Multiple microservices are often composed to create a combined functionality. However, the composition can be invalid.

4.1 Incompatible Components. Components can be incompatible, e.g., because one component sends data in the XML format and the other expects a JSON format (syntactic incompatibility), or e.g., one component sends a value that should be interpreted as dollars and the other expects a value in euros (semantic incompatibility).

4.2 Parts of Composition Missing. E.g., an authentication service that is needed to successfully use the services is missing.

4.3 Combined Functionality Faulty. When multiple services provide a combined functionality but it was incorrectly implemented. E.g., inter-connected data should be deleted from multiple services but is only deleted from some.

5. Connection Faults. This category considers faults that lie in the connection between two or more services, e.g., in the message queue or in the network that is used to transport messages from one service to another.

5.1 Connection Denied. According to our interviews, the connection to another service can be denied either due to a denied authorization (*Authorization Denied*), a failed authentication (*Authentication Failed*), or a *Wrong Configuration* of the connection.

5.2 Connection to Wrong Service. A service can try to connect itself to the *Wrong* (not intended) *Service* or a *Wrong Service Version* (one that is not compatible with the service). If a discovery mechanism is installed in the microservice system, this fault might manifest as a *Wrong Service Version* Discovery Fault. However, as a microservice system does not necessarily need to have a discovery component, the connection of two (or more) non-compatible services can also be a fault of the connection.

5.3 Timed Out. The connection can time out if, e.g., the connection component is not responding or is overloaded.

5.4 Connection Performance. The performance of the connection between two services can suffer either because the *Data Transmission* is *Too Slow*, *Too Much Data* is sent, or *Too Many Messages* are sent.

5.5 Connection Disruption. The connection between two services can be disrupted by the component that implements the connection (e.g., a message queue). This component can have its own *Time Restriction* or *Data Load Restriction*, which may have disrupting side effects on the connection between the services. The *Connection* itself can be *Interrupted* (e.g., through a failure in the network), or *Messages* can be *Lost* (e.g., due to a faulty message queue).

6. Execution Faults. Execution faults occur when two or more services are executed, but the result does not match the expected outcome.

6.1 Incorrect Result. The execution gives back an incorrect result, either because the input was incorrect (*Incorrect Input*), the data that the other service (instance) had stored and used for the execution was incorrect (*Service State Faulty*), or the other service was faulty (*Service Faulty*). An incorrect result can also happen because both services have an *Incompatible Context Interpretation*. A typical example is services that are executed in different time zones and, in consequence, interpret time data differently. In, e.g., a REST-API, the result code of the response can also be incorrect (*Wrong Result Code*), and the resulting data can be incomplete (*Data Incomplete*), e.g., due to pagination.

6.2 Timed Out. The service execution can time out because the other *Service Crashed* either *temporarily* or *permanently*, because of a *Communication Failure*, or because the other *Service* is *Too Slow*.

6.3 Incompatible Interfaces. The services' interfaces can be incompatible, resulting in either a *syntactic* or *semantic Conversation Fault*. An example of syntactic incompatibility would be when one service sends data in the XML format while the other expects data in the JSON format. With a semantic conversation fault, the interpretation of the exchanged data differs between the services. For example, one interprets an int value in dollars while the other interprets it in euros. The interfaces of two or more services can also be incompatible because one service sends more data than the other service can handle (*Too Much Data*).

6.4 Disruption by Environment. The execution of a request can also be disrupted by the environment (e.g., hardware components that stop working or do not work as intended).

6.5 Execution Performance. The interviewees described performance problems with the execution of requests that involved multiple services. In contrast to the *Service Too Slow* fault, this did not completely hinder the system's functionality but only resulted in a bad user experience.

B. Validation Results

The results of the validation survey are summarized in Table I. For each fault category, we report the percentage of participants who have encountered such a fault in one of their projects before ("Yes") and those who did not ("No"). We furthermore show the perceived severity of the faults and the perceived effort to fix such faults in a microservice system.

TABLE I
SURVEY RESULTS

Fault Category	Sub Category	Response		Severity			Effort to Fix		
		Yes	*No*	*Low*	*Moderate*	*High*	*Low*	*Moderate*	*High*
Description Fault	Format Fault	75%	25%	44%	19%	13%	56%	19%	0%
	Content Fault	88%	13%	19%	44%	19%	19%	44%	25%
	Service/ Description Mismatch	88%	13%	0%	50%	31%	0%	38%	50%
Service Deployment Fault	Required Resource Missing	94%	6%	13%	25%	56%	44%	44%	6%
	Wrong Configuration	100%	0%	19%	6%	75%	56%	13%	31%
	Service/Environment Incompatible	56%	44%	0%	6%	50%	13%	6%	38%
	Deployment Timed Out	88%	13%	25%	19%	44%	31%	25%	31%
Discovery Fault	No Service Found	69%	31%	6%	25%	38%	19%	31%	19%
	Wrong Service Found	19%	81%	0%	0%	19%	0%	13%	6%
	Routing Faulty	50%	50%	13%	19%	19%	31%	25%	6%
Composition Fault	Incompatible Components	94%	6%	13%	50%	31%	19%	31%	44%
	Parts of Composition Missing	50%	50%	0%	6%	44%	19%	13%	19%
	Combined Functionality Faulty	88%	13%	0%	44%	38%	0%	19%	63%
Connection Fault	Connection Denied	88%	13%	13%	44%	3%	31%	31%	19%
	Connection to Wrong Service	19%	81%	0%	0%	19%	6%	6%	6%
	Connection Timed Out	94%	6%	19%	50%	25%	19%	38%	38%
	Connection Performance	88%	13%	0%	44%	44%	0%	6%	81%
	Connection Disruption	69%	31%	6%	13%	50%	6%	31%	31%
Execution Fault	Incorrect Result	75%	25%	6%	38%	31%	6%	50%	19%
	Execution Timed Out	88%	13%	25%	38%	25%	0%	25%	56%
	Incompatible Interfaces	69%	31%	6%	6%	56%	6%	31%	31%
	Disruption by Environment	69%	31%	0%	13%	56%	19%	19%	31%
	Execution Performance	94%	6%	0%	56%	38%	6%	0%	88%

There is no fault category that none of the participants have ever encountered in their projects, which confirms that all the categories in the taxonomy are relevant. Most of the fault categories were experienced by at least 50% of the participants before. The most approved fault category is *Wrong Configuration* with 100% of "Yes" answers. Only the two fault categories *Wrong Service Found* and *Connection to Wrong Service* were encountered by less than 50% but still by nearly 20% of all participants. *Wrong Configuration* was perceived as the most critical regarding the severity, with 75% of all participants indicating the severity as "High". The fault category *Execution Performance* was indicated with the highest effort to fix as 87% of participants perceived the effort as "High". Some participants suggested missing faults in the taxonomy. One relates to a process fault, but its system consequences are covered. Another increases workload without causing system errors, so it is excluded. The remaining five fit under "Required Resource Missing", "Combined Functionality Faulty", and "Too Much Data". We believe participants could not find the correct category because the survey descriptions lacked examples specific to their experiences. Those results give us confidence that we did not miss any significant faults in our taxonomy. Additionally, given that most fault categories were experienced before by at least 50% of all participants, we conclude that the fault categories in our taxonomy are relevant.

IV. DISCUSSION WITH RELATED WORK

In the following, we discuss our results and compare our taxonomy with integration faults taxonomies and taxonomies for (micro-)service systems in related work.

A. Final Taxonomy vs. Integration Fault Taxonomies

A vast amount of work regarding recurring faults for different system types exists in the literature. Some of these works include integration-relevant faults considering various aspects of integration, among them integration faults in monolithic (e.g., [7]), cyber-physical (e.g., [9], [30]), or object-oriented (e.g., [10]) systems. Those taxonomies are insufficient for the microservice context, as they do not include aspects that come with the infrastructure and distributed nature of microservice systems. Our taxonomy includes a multitude of fault categories that relate to faults in the deployment of services, discovery mechanisms, or connection components, which can not be found in those taxonomies for other system types.

Taxonomies with faults in the composition of SOA services (e.g., [11], [12]) and web services (e.g., [31]) do include some faults that are also important for the microservice context, as, e.g., one service not being able to find a specific another service. However, unlike our work, they are not based on empirical research. Therefore, it is unclear how realistic the faults in these taxonomies are. To the best of our knowledge, our work is the first integration-relevant fault taxonomy for microservice systems that includes the experience and knowledge from practice.

B. Final Taxonomy vs. Fault Taxonomies for (Micro-)Service Systems

Several fault taxonomies exist for SOA systems [11], [12], [32], with one specifically tailored for microservice systems [13]. The SOA fault taxonomy proposed by Bruning et al. [11] served as our starting point in developing the taxonomy presented in this paper. However, it is important to note a significant contrast: Bruning et al.'s [11] taxonomy lacks an empirical foundation and is not designed for microservice systems, which is in contrast to ours, which specifically targets this context.

Building upon the work of Bruning et al. [11], Bhandari and Gupta [12] extended the taxonomy but remained within the domain of SOA systems. As we have described in more detail in Sec. II-B, there are differences between "textbook" SOA systems and microservice systems, which also manifests in the fault taxonomies. They conducted a systematic literature analysis to gather knowledge about faults in SOA systems from the literature. In contrast, we conducted interviews with practitioners to get a more practical point of view.

Marculescu et al. [32] analyzed and classified the faults found in REST APIs of web services by automated test generation with EvoMaster [33]–[35]. Their taxonomy focuses on faults in REST APIs, while we opted to include different communication mechanisms as well (e.g., message queues). Additionally, they focus on all faults found through automatic test generation while we focus on integration-relevant faults without a specific test creation strategy.

To the best of our knowledge, the taxonomy by Silva et al. [13] is the only taxonomy of faults in microservice systems. However, it focuses on the impact of faults on non-functional attributes, such as maintainability or performance. Despite this contribution, the taxonomy poses challenges for usage in the context of testing, e.g., creating a systematic fault injection approach or test strategies, as many aspects of testing are rather focused on functional attributes. Additionally, their taxonomy is based on 28 studies from academia and 3 from grey literature. Our work instead emphasizes insights from the industry through interviews and surveys with practitioners. Moreover, our taxonomy revolves around integration-relevant faults, making it suitable for evaluating or creating integration-level tests and test strategies within a microservice system.

In the following, we will compare the two taxonomies in more detail: Silva et al. [13] group their fault categories into six different categories: performance, security, reliability, maintainability, compatibility, and functionality. The categories with the most fault types are maintainability, security, and performance. Unfortunately, Silva et al. only explain the higher level categories regarding non-functional attributes but do not describe the individual fault types in their paper. However, they provided a replication package [36], that does include descriptions of the examples that lead them to form those fault types. Therefore, we used those descriptions for our comparison and will quote excerpts of those directly for ease of comprehensibility.

Maintainability Faults. Many of their maintainability faults, e.g., "Invalid User Input Fault", "Missing User Input Fault", "Expired Request Data Fault", and "Invalid Parameter Query" align or fall under our broader "Incorrect Input" category. Some of those fault categories are highly specific to certain platforms or technologies. E.g., "Invalid Parameter Query" pertains to AWS query strings, while "Incomplete Signature" relates to AWS signature mechanisms. We have intentionally opted for a higher level of abstraction in our taxonomy to maximize its comprehensibility and adaptability across diverse microservice environments rather than tying them to specific technologies.

Security Faults. Most security-related faults such as "Insecure Data Exposure", "Missing Authorization", and "Missing Authentication" are not represented in our taxonomy. This is where the key difference between our two taxonomies becomes apparent: while Silva et al. focused on non-functional attributes, we focus on functionality. Therefore, we only consider faults where existing authorization or authentication mechanisms malfunction, which are represented by the categories "Authorization Denied" and "Authentication Failed". The complete absence of authorization or authentication mechanisms in a system lies outside of our testing focus.

Performance Faults. We did not set a clear focus on performance faults for our taxonomy, yet they can be found in multiple parts. E.g., a time out in the connection to another service or just a bad connection performance. Still, Silva et al. provide much more fault types here. Many of those are not relevant to the integration context, among them "CPU Allocation Fault", "Memory Allocation Fault", and "Memory Usage Fault". Those are faults in single microservices that should already be targeted and detected during unit or component testing and not during testing the integration of multiple services. Therefore, they are not relevant to our taxonomy of integration faults. Fault categories relevant to the integration of microservices (as they either lie in the connection or affect the communication between services), such as "Increased message size", or "Increased number of Users" can also be found in our taxonomy: "Too Much Data" and "Too Many Messages".

Functional Faults. Silva et al.'s "Functional Faults" category only consists of two different fault types, namely "Functional Fault" and "Internal Fault". They describe the "Functional Fault" as *"Result in malfunctioning of system services by raising errors or producing incorrect results"*, which is relatively vague. Our taxonomy includes multiple different sources/reasons why a service might produce an incorrect result (see the six fault types under "Incorrect Result"). Here, again, the different focuses of those two taxonomies become prevalent. "Internal Fault" is described as *"The root causes of Internal faults lie in the internal implementation of individual microservices"*. Due to this definition, we consider this fault category as faults that should be targeted and detected during unit or component testing and not relevant for the integration testing context.

In summary, the overall structure and focus of the two

taxonomies are vastly different, as they were built for two different purposes: Silva et al.'s taxonomy was built to reflect the influence of fault types on different non-functional attributes. Our taxonomy was built to provide a structured overview of integration faults to facilitate the test strategy and test case creation of integration-level tests in microservice systems.

For many of the fault categories in our taxonomy, we could not find a matching fault type in Silva et al.'s taxonomy. We have two inner categories and eight leaf fault types in our "Service Description Faults" category. Yet, we could not find any indication of fault types regarding the description or specification of a service in Silva et al.'s work. Our taxonomy consists of four deployment fault types, out of which only two are represented in the taxonomy of Silva et al. Furthermore, our taxonomy holds six discovery faults, yet we could not find any faults in service discovery mechanisms in Silva et al.'s taxonomy. Two of our "Connection Faults" (regarding authorization and authentication) can also be found in Silva et al.'s taxonomy. We also found an example for a "Wrong Configuration" fault in their taxonomy, however, our fault type is more broad and does include more aspects than an *"[...] incorrect configuration of the API consumer account"*. For "Connection to Wrong Service" and "Connection Time Out", we could not find resembling fault types in the other taxonomy. Our four "Connection Disruption" faults fall under their "API Internal Fault" fault type. However, we provide four different root causes for such a fault in contrast to them. As already mentioned above, we also provide six different root causes for when the result of a request to another service is incorrect, whereas Silva et al. summarize this under one, broader fault type, namely "Functional Fault". We also provide six more fault types (two of them having two more specializations each) that can occur during the execution of a request to another service which we could not find in Sila et al.'s taxonomy.

C. Summary

Our proposed taxonomy fills a research gap, providing a structured overview specifically targeting integration-relevant faults in microservice systems and catering to the needs of microservice testing. In addition, we provide insights into the perceived severity and effort to fix individual fault categories that are not provided in existing literature. This information can further facilitate the prioritization of specific fault categories when creating test strategies or test cases, as well as testing the robustness of a system, which can help to allocate resources more efficiently. Therefore, our new taxonomy provides valuable insights and contributions in the context of microservice integration testing.

V. Threats to Validity

Internal Validity. A threat to the validity of the taxonomy is biased coding of the interview transcripts. To mitigate this threat, each of the interviews was labeled by two evaluators. Additionally, it is possible that questions asked during the interviews might have been affected by the initial taxonomy for SOA systems. However, we have intentionally kept the questions as generic as possible, following a generic framework for the elicitation of defects proven effective before [24]. Furthermore, in our validation study, we did not fully validate the structure of the taxonomy. This was a conscious choice as we needed to keep the effort of the survey to a manageable time frame to acquire as many participants as possible. As we did not invent the structure of the taxonomy but used the structure of an existing taxonomy for SOA systems that is well-established, we do not see this as a great limitation to our validation study. Additionally, to make sure that our first adaptation of the original SOA taxonomy into the microservice context is correct, we presented preliminary adaptations to the microservice community [22] and incorporated their feedback into our final adaptations. Finally, the possibility that survey participants might have misinterpreted the fault categories poses a significant threat to our validation. To mitigate this threat, we conducted multiple test runs of the survey and the fault descriptions with participants who were familiar with the overall topic and practitioners in the same field. We used their feedback to improve the fault and survey descriptions.

External Validity. As microservice-based systems can be used in many different domains or setups, including mere software systems, connections to hardware, legacy systems, etc., it is hard to decide whether our results are generalizable to all other microservice systems. An additional potential threat to external validity arises from the limited number of interview participants, as the findings are based on insights from only ten individuals. However, to get diverse perspectives, we interviewed practitioners from multiple companies with a combined experience from more than 20 projects in 9 different project domains.

VI. Conclusion

We have built a taxonomy of integration-relevant faults in microservice systems based on an existing taxonomy for SOA and interviews with 10 practitioners. The taxonomy consists of 6 main categories containing 61 lower-level fault categories on up to three levels of detail. To validate our taxonomy and enrich it with information regarding the perceived severity and effort of the respective fault categories, we performed a survey with 16 practitioners. The results of this validation survey showed that most of the fault categories (21/23) were experienced by at least 50% of all participants in at least one project before.

To the best of our knowledge, this is the first fault taxonomy focusing on integration-relevant faults in microservice systems including insights from practice. The resulting taxonomy can be used for the systematic identification and classification of faults in microservice systems, test case design, risk assessment, improving the communication between different development teams, and stakeholders, enhancing software reliability through fault injection approaches, and to create new mutation operators.

VII. Data -Availability statement

To strengthen transparency and facilitate replication, we provide a replication package [20].

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

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

[8] B. Beizer, *Software system testing and quality assurance*. Van Nostrand Reinhold Co., 1984.

[9] A. M. Madni and M. Sievers, "Systems integration: Key perspectives, experiences, and challenges," *Systems Engineering*, vol. 17, no. 1, pp. 37–51, 2014. DOI: https://doi.org/10.1002/sys.21249.

[10] M. Winter, M. Ekssir-Monfared, H. M. Sneed, R. Seidl, and L. Borner, *Der Integrationstest: Von Entwurf und Architektur zur Komponenten-und Systemint.* 2012.

[11] S. Bruning, S. Weissleder, and M. Malek, "A Fault Taxonomy for Service-Oriented Architecture," in *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, ISSN: 1530-2059, Nov. 2007, pp. 367–368. DOI: 10.1109/HASE.2007.46.

[12] G. P. Bhandari and R. Gupta, "Extended Fault Taxonomy of SOA-Based Systems," en, *CIT. Journal of Computing and Information Technology*, vol. 25, no. 4, pp. 237–257, Jan. 2018, Number: 4. DOI: 10.20532/cit.2017.1003569.

[13] 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.

[14] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 510–520. DOI: https://doi.org/10.1145/3338906.3338955.

[15] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE transactions on software engineering*, vol. 32, no. 4, pp. 240–253, 2006. DOI: 10.1109/TSE.2006.38.

[16] Z. Long, G. Wu, X. Chen, C. Cui, W. Chen, and J. Wei, "Fitness-guided Resilience Testing of Microservice-based Applications," in *2020 IEEE International Conference on Web Services (ICWS)*, Oct. 2020, pp. 151–158. DOI: 10.1109/ICWS49710.2020.00027.

[17] K. Meinke and P. Nycander, "Learning-Based Testing of Distributed Microservice Architectures: Correctness and Fault Injection," in *Software Engineering and Formal Methods*, ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, pp. 3–10. DOI: 10.1007/978-3-662-49224-6_1.

[18] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic Resilience Testing of Microservices," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2016, pp. 57–66. DOI: 10.1109/ICDCS.2016.11.

[19] N. Humbatova, G. Jahangirova, and P. Tonella, "Deepcrime: Mutation testing of deep learning systems based on real faults," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021, pp. 67–78. DOI: 10.1145/3460319.3464825.

[20] Replication Package., 2025. DOI: 10.6084/m9.figshare.27046156.v1.

[21] J. Hu, I. Khalil, S. Han, and A. Mahmood, "Seamless integration of dependability and security concepts in SOA: A feedback control system based framework and taxonomy," en, *Journal of Network and Computer Applications*, vol. 34, no. 4, pp. 1150–1159, Jul. 2011. DOI: 10.1016/j.jnca.2010.11.013.

[22] L. Gregor, A. Pretschner, A. Hentschel, H. Sauer, and M. Saft, "The Fault in Our Services: Investigating Integration-Relevant Faults in Microservices," 2023.

[23] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study:" in *Proceedings of the 8th International Conference on Cloud Computing and Services Science*, SCITEPRESS - Science and Technology Publications, 2018, pp. 221–232. DOI: 10.5220/0006798302210232.

[24] D. Holling, D. M. Fernández, and A. Pretschner, "A Field Study on the Elicitation and Classification of Defects for Defect Models," en, in *Product-Focused Software Process Improvement*, ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2015, pp. 380–396. DOI: 10.1007/978-3-319-26844-6_28.

[25] S. Hove and B. Anda, "Experiences from conducting semi-structured interviews in empirical software engineering research," in *11th IEEE International Software Metrics Symposium (METRICS'05)*, 2005, 10 pp.–23. DOI: 10.1109/METRICS.2005.24.

[26] V. Ligurgo, T. Philippette, P. Fastrez, A.-S. Collard, and J. Jacques, "A method combining deductive and inductive principles to define work-related digital media literacy competences," in *Information Literacy in the Workplace*, Cham: Springer International Publishing, 2018, pp. 245–254. DOI: https://doi.org/10.1007/978-3-319-74334-9_26.

[27] B. F. Crabtree and W. F. Miller, "A template approach to text analysis: Developing and using codebooks," in *Doing qualitative research*, ser. Research methods for primary care, Vol. 3. Thousand Oaks, CA, US: Sage Publications, Inc, 1992, pp. 93–109.

[28] R. E. Boyatzis, *Transforming qualitative information: Thematic analysis and code development*. sage, 1998.

[29] D. R. Thomas, "A general inductive approach for analyzing qualitative evaluation data," *American journal of evaluation*, vol. 27, no. 2, pp. 237–246, 2006.

[30] N. G. Leveson, "Role of software in spacecraft accidents," *Journal of spacecraft and Rockets*, vol. 41, no. 4, pp. 564–575, 2004.

[31] K. S. M. Chan, J. Bishop, J. Steyn, L. Baresi, and S. Guinea, "A Fault Taxonomy for Web Service Composition," in *Service-Oriented Computing*, 2009, pp. 363–375. DOI: 10.1007/978-3-540-93851-4_36.

[32] 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.

[33] A. Arcuri, "Evomaster: Evolutionary multi-context automated system test generation," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2018, pp. 394–397. DOI: https://doi.org/10.1109/ICST.2018.00046.

[34] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, pp. 1–37, 2019. DOI: 10.1145/3293455.

[35] A. Arcuri, J. P. Galeotti, B. Marculescu, and M. Zhang, "Evomaster: A search-based system test generation tool," 2021.

[36] Replication Package for "Towards a Fault Taxonomy for Microservices-Based Applications". https://github.com/Gutenbergf/Fault-Taxonomy-for-Microservice-Based-Applications, 2022.