*Journal of Internet Services and Applications, 2024, 15:1,* doi: 10.5753/jisa.2024.4000


# Dependable Microservices in the Kubernetes era: A Practitioners Survey

**Vinícius J. S. Souza** [ **Universidade Federal de São Paulo (UNIFESP)** | ***vjssouza@unifesp.br*** ]
**Vânia O. Neves** [ **Universidade Federal Fluminense (UFF)** | ***vania@ic.uff.br*** ]
**Bruno Y. L. Kimura** [ **Universidade Federal de São Paulo (UNIFESP)** | ***bruno.kimura@unifesp.br*** ]

*Universidade Federal of São Paulo, Av. Cesare Mansueto Giulio Lattes, 1201, São José dos Campos - SP, Brazil.*

**Received:** 31 December 2023 • **Accepted:** 15 November 2024 • **Published:** 14 December 2024

**Abstract** The microservices architectural style offers several advantages to software development, including independence among development teams, greater autonomy for developers, faster product development, and improved scalability. However, since the communication topology relies on distributed systems, faults become more frequent and harder to manage, posing challenges to reliability and availability, which are key attributes of business-critical services. To address these concerns, fault patterns, countermeasures, and technologies have been explored and implemented in both industry and academia to prevent, tolerate, mitigate, and predict faults in microservices. To understand current industry practices for achieving dependable microservices, we present the results of an opinion survey with microservice practitioners, aiming to identify the main fault and failure patterns, countermeasure techniques, supporting technologies, existing gaps, and the evolution of the field. We also provide a review of academic research in this area, examining the connections between industry practices and academic literature, highlighting key findings, challenges, and opportunities.

**Keywords:** Microservices, Dependability, Faults, Failures, Countermeasure Techniques, Technologies.

## 1 Introduction

In modern software systems, single executable artifacts, known as monoliths, have increasingly been migrated to a microservices-based architecture. In this architecture, the application is divided into several cohesive services, each designed to fulfill a specific subset of responsibilities within the overall system [Dragoni *et al*., 2017]. Specifically, a microservice is an independent process that provides functionality to other microservices over the network, typically using remote procedure calls or messaging [Fowler, 2014]. A system comprises self-contained services with well-defined boundaries, which are cohesive enough to be managed and maintained by a small team of developers. The microservice architectural style offers several benefits, including faster product development, enhanced scalability for users and developers, and greater autonomy for development teams [Jamshidi *et al*., 2018].

Despite the benefits, microservices present certain challenges. These challenges include increased system complexity and multiple potential sources of faults arising from inter-process communication over a network. As a result, managing faults in microservices effectively becomes a complex task [Jamshidi *et al*., 2018]. Ensuring system dependability is crucial to addressing these challenges. In particular, two attributes are critical: *Availability*, defined as the proportion of time the application is operational and able to receive requests [Hammer, 2007]; and *Reliability*, which is the probability that a software component will function as specified over a given period [Zo *et al*., 2007].

Research efforts on microservices have expanded significantly in recent years[1]. Architectural patterns and principles designed to enhancing the dependability of microservices have been applied in industry [Richardson, 2019] and cataloged [Taibi *et al*., 2018]. These patterns have been utilized to enhance the quality and reliability of microservices APIs [Stocker *et al*., 2018]. Studies have examined failures to identify their root causes across various contexts, including the web [Padmanabhan *et al*., 2006], data centers [Gill *et al*., 2011], and cloud environments [Jhawar and Piuri, 2017; Potharaju and Jain, 2013]. Additionally, tools and methods for monitoring microservices have been proposed [Heger *et al*., 2017; Haselböck and Weinreich, 2017]. Verification techniques, such as formal methods and chaos engineering [McCaffrey, 2015; Panda *et al*., 2017], have also been adapted and applied to the microservices domain.

As the microservices architectural style continues to gain relevance in the industry [Thönes, 2015], analyzing best practices for developing dependable microservices is of interest to both industry and academia. This analysis helps identify challenges and opportunities. These insights can inform future research on dependable microservices. To this end, in this study, we conducted an opinion survey with microservice industry practitioners, supported by a literature review. An opinion survey gathers knowledge from individuals to understand specific aspects of a population [Wohlin *et al*., 2012], while a literature review collects insights from academic research and evaluates gaps identified in the industry. Particularly, our goal is to answer the following questions:

[1] According to Google Scholar, more than 8,000 studies on microservices have been published annually since 2021, and over the last decade, this number exceeds 38,000.

RQ1: *How is the industry dealing with dependability on microservices, in terms of experienced faults and failures, countermeasures applied (techniques and technologies)?*
RQ2: *What are the main gaps and difficulties in terms of making microservices more dependable?*
RQ3: *What related topics have been researched by academia in relation to the opinion survey? How much are they correlated with industry gaps?*

The contributions of this work are highlighted as follows. We present the results of an opinion survey conducted with industry practitioners in 2024, analyzing how faults and failures are perceived, the techniques adopted to address them, the supporting technologies used, the methods for diagnosing faults and failures, and the difficulties faced in the diagnostic process. We review the literature to identify academic contributions in this area, focusing on works published between 2020 and 2024. This review enables a comparison between the gaps identified in the industry and the related topics explored in academia. Finally, we discuss the key findings, challenges, and opportunities by connecting the survey results with the insights from the literature review.

The opinion survey reveals that professionals are increasingly adopting advanced technologies, such as load balancing, health checks, and the `Kubernetes` container orchestrator. However, advanced techniques for fault removal and prediction remain underutilized. In contrast, the literature review highlights a strong academic focus on automatic diagnosis and fault injection techniques as countermeasures to enhance the dependability of microservices-based architectures. This academic trend demonstrates a clear understanding of industrial challenges and a commitment to developing practical solutions. Nevertheless, there remains a gap between the theoretical approaches investigated in academia and their practical application in the industry, indicating a divergence between academic advancements and the effective use of these technologies by practitioners.

The remainder of this paper is organized as follows. In the next section, we discuss related works. In Section 3, we present the background on microservices and dependability. Section 4 describes the methodology adopted to conduct the opinion survey and the literature review. In Section 5, we quantitatively analyze the main results of the survey. Section 6 presents the results of the literature review. In Section 7, we discuss the key findings, challenges, and opportunities. Section 8 addresses the threats to validity of both the practitioners' survey and the literature review. Finally, Section 9 brings our main conclusions.

## 2 Related work

A number of studies have conducted opinion surveys with industry practitioners to understand different aspects of microservice architecture and to gain insights into development practices. Some of these studies have focused on specific activities during the development process such as migrations [Ghofrani and Bozorgmehr, 2019; Di Francesco *et al*., 2018]. Others have provided a more general overview of the state of practice [Viggiato *et al*., 2018; Bogner *et al*., 2019; Knoche and Hasselbring, 2019; Ghofrani and Lübke, 2018]. Among the studies, Viggiato *et al*. [2018] conducted a quantitative survey with 122 professionals to understand how the industry uses microservices architecture and verify whether the perception of microservices advantages and challenges is in line with the literature. Similarly, Bogner *et al*. [2019] interviewed 17 professionals from 10 companies in Germany to explore the applied technologies, characteristics of microservices, and thier perceived influence on software quality.

Knoche and Hasselbring [2019] considered responses from 71 German participants to gain insights into the reasons why companies are considering the adoption of microservices. The authors investigated to what extent microservices were perceived as a tool for software modernization, what objectives are sought when introducing microservices into existing software, and how the potential impact on runtime performance and transactionality is evaluated.

Ghofrani and Lübke [2018] investigated the main challenges and concerns in designing and developing microservices-based systems, as well as the key reasons for leveraging and hindering the use of systematic approaches in a microservices architecture. The authors also reported that resilience, reliability, fault tolerance, and memory usage are essential points to consider during microservices development.

Zhou *et al*. [2018] conducted an industrial survey on fault analysis and debugging microservices applications, incorporating a benchmark application for experimental validation. The survey findings indicate that the majority of identified faults are functional, leading to incorrect outcomes, runtime failures, or lack of response. Conversely, a smaller proportion of faults are non-functional, which contributes to unreliable services or prolonged response times. The survey also reveals that all participants consistently rely on log analysis as the primary method for both fault analysis and debugging.

The authors classified the debugging practices and techniques into three levels of maturity: First, Basic Log Analysis, where developers manually sift through extensive log data to locate faults, with success heavily dependent on their familiarity with the system. The second is Visual Log Analysis, which employs tools to collect, retrieve, and visualize logs. Third, Visual Trace Analysis involves the use of advanced tools for microservice execution tracing and visualization.

Soldani and Brogi [2022] present the results of a literature review on existing techniques for anomaly detection and root cause analysis in modern multi-service applications. They classify the techniques into three main categories: log-based, distributed tracing, and monitoring. Most techniques rely on processing data collected during training runs of the applications, using machine learning to create baseline behavior models. Additionally, some techniques already integrate anomaly detection and root cause analysis into a single pipeline, while others require additional integration to interoperate. The study also identifies open challenges in anomaly detection, such as the need to improve accuracy and reduce configuration costs.

Nasab *et al*. [2023] accomplished an empirical study in which 28 security practices were identified by analyzing se-

| Work | System Type | Focus Area | Study Type | Participants | Year |
|---|---|---|---|---|---|
| [Viggiato *et al*., 2018] | Microservices | Overview | Opinion Survey | 122 | 2018 |
| [Bogner *et al*., 2019] | Microservices | Overview | Interviews | 17 | 2019 |
| [Knoche and Hasselbring, 2019] | Microservices | Overview | Opinion Survey | 71 | 2019 |
| [Ghofrani and Lübke, 2018] | Microservices | Overview | Opinion Survey | 25 | 2018 |
| [Nasab *et al*., 2023] | Microservices | Security | Opinion Survey | 63 | 2023 |
| [Ghofrani and Bozorgmehr, 2019] | Microservices | Migration | Interviews | 17 | 2019 |
| [Di Francesco *et al*., 2018] | Microservices | Migration | Interviews, Opinion Survey | 5 and 18 | 2018 |
| [Zhou *et al*., 2018] | Microservices | Fault Analysis, Debugging | Interviews | 16 | 2021 |
| [Waseem *et al*., 2021] | Microservices | Design, Monitoring, Testing | Opinion Survey, Interviews | 106 and 6 | 2021 |
| [Niedermaier *et al*., 2019] | Distributed Systems | Observability, Monitoring | Interviews | 28 | 2019 |
| [Wong *et al*., 2016] | Service-oriented Systems | Fault Localization Techniques | Literature Survey | N/A | 2016 |
| [Łgorzata Steinder and Sethi, 2004] | Communication Systems | Fault Localization Techniques | Literature Survey | N/A | 2004 |
| [Soldani and Brogi, 2022] | Services and Microservices | Fault Localization | Literature Survey | N/A | 2022 |
| [Amiri *et al*., 2023] | Distributed Systems | Resiliency and Dependability | Literature Survey | N/A | 2023 |
| **This survey** | Microservices | Dependability | Opinion Survey | 46 | 2024 |

**Table 1.** A qualitative comparison with related work.

curity points from GitHub repositories and Stack Overflow posts. The authors organized the practices into six groups: Authorization and Authentication, Token and Credentials, Internal and External Microservices, Microservices Communications, Private Microservices, and Database and Environments. Based on this collection, they conducted a survey with microservices practitioners to evaluate the practices' effectiveness, concluding that most of them are recommended for use in the industry.

Waseem *et al*. [2021] discussed results of a survey with 106 professionals and interviews with six experts in microservices, aiming to gain a deep understanding of the design, monitoring, and testing of microservice architectures in practice. The survey results include the main practices and metrics used to monitor microservices and main design patterns applied. The study highlights several significant findings. Regarding monitoring, the most commonly used metrics to evaluate microservices systems include resource usage, load balancing, and system availability. Among the most adopted monitoring practices are log management, exception tracking, and the use of health check APIs. The research also identifies crucial challenges in monitoring, such as the collection of metrics and logs in container environments and the complexities of distributed tracing. In terms of testing, the methods most utilized by participants include unit testing, end-to-end (E2E) testing, and integration testing. The main challenges are primarily related to manual testing, integration testing, and debugging microservices deployed on container platforms.

Other works investigate the reliability of distributed systems more broadly. Amiri *et al*. [2023] presents a systematic review of resilience and reliability in distributed systems, with a special focus on the Internet of Things (IoT) and its extensions, such as the Internet of Drones (IoD) and the Internet of Vehicles (IoV). It introduces a new taxonomy that classifies distributed environments into seven main categories: cloud, edge, fog, IoT, IoD, IoV, and hybrid systems. The analysis of 37 articles highlighted that security, latency, and fault tolerance are the most frequently encountered parameters, playing crucial roles in the resilience management of these environments.

Niedermaier *et al*. [2019] carried out a study with industry about observability and monitoring of distributed systems, showing the available techniques, methods, and tools. Wong *et al*. [2016] present a survey on software fault localization techniques, including works on service-oriented systems. In the same direction, Łgorzata Steinder and Sethi [2004] presented a survey on fault localization techniques on communication systems, classifying the solutions proposed in the last ten years.

Table 1 summarizes a qualitative comparison among related works. While several studies have investigated relevant aspects of microservices architecture and development practices, to the best of our knowledge, there is no other study that has taken a deeper understanding of the dependability of microservices. In this context, this work attempts to fill the gap by providing an overview of the state of practice regarding microservices dependability. It covers all the means to ensure it, i.e., fault prevention, tolerance, removal, and forecasting, instead of restricting to a specific aspect. The goal is not to introduce new patterns, techniques, or technologies but rather to collect and discuss existing ones based on the results of an opinion survey conducted with industry practitioners and a literature review.

# 3 Background

We present in the following the main concepts on microservices and on software dependability.

## 3.1 Monoliths and microservices

An architectural style is a set of rules for developing applications, organizing components and interactions, and manipulating data [Kumar, 2014]. Monolith and microservices are, thus, architectural styles [Dragoni *et al*., 2017], specifically:

- *Monolith*: an application is decomposed into modules that cannot be executed independently, while being encapsulated into a single executable artifact.
- *Microservice*: in contrast, the application is decomposed into independent artifacts in terms of development, building, and deployment.

| Interaction | Model | Style | Abstraction | Data Format |
|---|---|---|---|---|
| Synchronous | UC | REST, RPC | HTTP(S) | JSON, XML |
| Synchronous | UC | RPC | gRPC | protocol buffers |
| Asynchronous | UC, MC | Message | Broker | text, binary |

**Table 2.** Examples of communication attributes for microservices.

In fact, a monolith can be decomposed into microservices, reducing the size of the application according to the desired scalability and the organization of developers' teams [Newman, 2019]. While there is no consensus about the ideal size of a microservice, its granularity potentially range from a few dozen to thousands of lines of code [Jamshidi *et al*., 2018]. Differently from monolith modules, that usually share resources (e.g., database, memory, file system), a microservices-based system is distributed, with smaller, independent applications communicating through message exchange. External requests can be received at a centralized entity, typically an API gateway [Taibi *et al*., 2018], which forward requests to the target microservices. Multiple replicas of microservices can run concurrently to divide the workload over the replicas. When migrating from a monolith to microservices, splitting the database into smaller instances is also practice, preventing access serialization and conflicts, while requiring synchronizing data between the databases instances to avoid inconsistency [Newman, 2019]. Another practice, although not very recommended due to synchronization complexity, is sharing the database instances between monolith and microservices until the migration is finished [Taibi *et al*., 2018].

## 3.2 Communication models

Microservices usually communicate using *request-reply* or *publish-subscribe* interaction styles. A request allows a microservice to use features provided by other microservices. Such a communication can be accomplished with different attributes [Richardson, 2019]: interaction mode can be synchronous or asynchronous; traffic model can be unicast (UC), for a transmission between a single pair of microservices (i.e., point-to-point), or multicast (MC), for multiple destinations (e.g., in a publish-subscribe pattern). Table 2 describes common communication attributes for microservices architectures.

### 3.2.1 Styles

Remote Procedure Call (RPC) allows transparent point-to-point and synchronous communications, similar to local procedure calls, while being possible to adapt it to asynchronous and multicast transmissions [Tanenbaum and Van Steen, 2007]. Representational State Transfer (REST) architectural style uses a remote, synchronous and point-to-point interaction. While RPC focuses on the definition of actions, REST focuses on the business entities, modeling them as web resources, with actions accomplished through Hypertext Transfer Protocol (HTTP) methods [Newman, 2015]. Although allowing transparency, both RPC and REST may not meet non-functional requirements, like availability (when the receiver is not available, the communication is lost). Alternatively, message-oriented communication enables asynchronous, non-blocking, point-to-point, and multicast communication [Tanenbaum and Van Steen, 2007]. Such communication style can increase system availability by handling requests asynchronously in background worker processes. Suitable for applications with no strict latency requirements, message-oriented communications allow sending notifications, handling application events, while executing batch jobs and distributed transactions [Richardson, 2019].

### 3.2.2 Abstractions

To encapsulate implementation details on top of *socket* descriptors, abstractions have been a common practice [Tanenbaum and Van Steen, 2007]. For synchronous mode, HTTP [Fielding and Taylor, 2000] is widely used to provide such an encapsulation with persistent, secure, and reliable connections over TCP [Mogul, 1995]. Recently, greater performance with multiplexed requests, data compression, and binary headers are allowed with HTTP/2.0 [Belshe *et al*., 2015]. For asynchronous mode, a central entity, namely message broker, is responsible for validating, transforming and forwarding messages, while providing a communication infrastructure for distributed processes [Rostanski *et al*., 2014]. However, as it acts as an intermediate node, a broker increases end-to-end response time. Common broker implementations are `Apache Kafka` [Apache, 2023], `RabbitMQ` [VMware, 2023], and `AWS Kinesis` [Amazon, 2020].

### 3.2.3 Data formats

JSON and XML have been the most popular text-based formats [Sill, 2016]. Due to standardization, these formats are independent of the communication protocol and widely supported by programming languages. However, their text verbosity leads to overhead, more bandwidth usage, and poor support for data types, e.g., large numbers, dates, and time [Kleppmann, 2017]. With better support for schema definition, XML introduces redundant tags, being more verbose than JSON [Maeda, 2011]. On the other hand, binary formats such as `Avro` [Apache, 2022] and Protocol Buffers [Protocol-Buffers, 2023] allow data compression. Instead of tagging data fields, binary formats combine an Interface Definition Language (IDL) and primitive data types to represent fields [Kleppmann, 2017].

## 3.3 Service meshes

Service discovery, traffic routing, security, and load balancing are system functionalities usually orthogonal to the microservices implementation. While enabling such functionalities into microservices would result in code duplication, *service meshes* can handle orthogonal responsibilities through an infrastructure that intermediates the communication between the microservices [Richardson, 2019].

## 3.4 Dependability in microservices

In this section, we describe microservice dependability from three dimensions to substantiate the opinion sur-

vey: (1) types of faults and failures, with general events that microservice-based systems are prone to be affected; (2) countermeasure techniques, with typical methods to overcome the common faults and failures; and (3) technologies from State-of-The-Art solutions that implement the countermeasure techniques.

### 3.4.1 Terminology

Failures, errors, and faults are key terminologies in this work. To distinguish each one, from [Avizienis *et al*., 2004], we assume that a *failure* impacts end-users (or other services) and it is caused by an incorrect system state, i.e., an *error*, while the cause of an error is a *fault*. If a microservice preserves its functionality in the presence of faults, it can *mask* or *tolerate* faults. A *transient* fault is intermittent in time, while a *permanent* fault is continuous. A *functional failure* is resulted from faults of specific application use cases, while a *non-functional failure* comes from application-independent faults, e.g., resource, communication [Zhou *et al*., 2018].

Failures can be classified according to their domain, detectability, and consistency. If a service stops producing responses, it is classified as *omission* failure, e.g., HTTP status code 503 (*service unavailable*) returned from a microservice. If a service does not produce responses within the specified deadline so it times out, then it is classified as *timing* failure. As the output produced by the service is incorrect, we call it a *content* or *response* failure, e.g. a HTTP service returning an internal error with status code 500. If a failure is not consistent, i.e., being perceived differently by different users, we call it *inconsistent* or a *Byzantine* failure. Finally, if the service stops responding until it is restarted, we call it a *crash* failure [Cristian, 1991].

### 3.4.2 Common types of faults and failures

**Communication faults.** Microservices rely on remote interprocess communication through the TCP/IP protocol stack. At the network infrastructure, i.e., lower layers (physical, link, or network), transient faults such as packet losses, bit errors, congested links, firewall filters, and packet drops are the most common ones, leading to connectivity loss [Potharaju and Jain, 2013]. At the transport layer, even using TCP to overcome network transient faults with reliable connections, microservice requests are exposed to connection errors, e.g., connection refused, timeouts, resource exhaustion in thread pools [Nygard, 2018; Al-Qudah *et al*., 2010; Padmanabhan *et al*., 2006]. Since common architectural styles (REST, RPC, message-oriented) depend on upper layer protocols such as HTTP and, hence, Domain Name System (DNS), microservice requests may fail at application layer as well, e.g., DNS timeouts, HTTP errors. Communication faults, if not masked, may cause timing, omission, and even Byzantine failures. If a microservice $A$ depends on a message sent to $B$ to execute an use case, a failure on the communication with $B$ will make unavailable that functionality provided by $A$.

**Hardware faults.** Although commodity hardware has a typical life span of a few years (3 to 5 years [Vishwanath and Nagappan, 2010]), in a datacenter with thousands of hardware components, the likelihood of failures is not negligible. One can mention [Vishwanath and Nagappan, 2010], 70% of failures come from hard drives, 6% from RAID controllers, 5% from memory. If not handled correctly by fault tolerance mechanisms at the upper layers, such faults might cause crash failures on microservices. Cloud providers usually offer limited fault-tolerance mechanisms, in most cases requiring specific actions or configuration by the system administrator. For instance, to recover a virtual machine (VM) from a hardware failure, the VM image must be reloaded on another instance. While such a process is not automatic, it can be automated through monitoring [Wittig *et al*., 2016]. In catastrophic scenarios of hardware failures, an entire data-center may become unavailable, so that an important measure would be using geographically distinct datacenters [Barr *et al*., 2011].

**Resources exhaustion faults.** One can mention [Blohowiak *et al*., 2016; Nygard, 2018]: CPU exhaustion due to traffic spikes, and frequent memory management (garbage collection); memory exhaustion caused by spikes in request traffic, memory leak, unbounded caches, unbounded data access; disk exhaustion, when keeping microservice log files without limit (e.g., file rotation); and connections pools exhaustion, when long-lived connections lock shared resources in time, keeping other threads blocked, hence, increasing system response time. CPU and pools exhaustion leads to timing and omission failures, as the service will not be able to respond to the requests within the expected time. Memory exhaustion causes crash failures, as the service will fail to allocate new areas until it is restarted. In this scenario, the operating system should choose which processes to interrupt so as to keep the system working, e.g, Linux "out-of-memory killer". While disk exhaustion is taking place, response failures are returned from any request that requires writing to disk.

**Cascading failures.** Such failures occur progressively from a sequence of faults in a feedback loop, increasing over time, affecting several parts of the system [Beyer *et al*., 2016]. To take place, such failures are propagated through integration points between microservices [Nygard, 2018]. For example, requests to a malfunctioning downstream service can back propagate failures to the calling service. Cascading failures are triggered by the other types of faults listed in this section.

### 3.4.3 Countermeasure techniques

Figure 1 presents the corresponding set of techniques for each common type. We organize and describe the techniques into four categories, according to the goals in Avizienis *et al*. [2004]: fault prevention, fault tolerance, fault removal, and fault forecasting. Table 3 summarizes the description of the techniques and their goals.

**Fault prevention.** It takes place along with the software development, aiming at eliminating failures before happening in the production environment [Avizienis *et al*., 2004]. To prevent cascading failures, common countermeasure techniques are [Nygard, 2018]: interrupting workload with circuit breakers [Homer *et al*., 2014]; defining logical application delimiters with bulkheads. To prevent both cascading

| Goal | Countermeasure | Description |
|---|---|---|
| Prevention | Timeouts | Timers to limit delay of events, e.g., request processing, resource sharing. |
| | Circuit breakers | Temporarily interrupt requests upon recurrent failures or unavailability of external microservices. |
| | Bulkheads | Logical delimiters to isolate and decouple service components. |
| | Rate Limiter | Limiting the amount of tasks (e.g., requests) a microservice can handle in a time interval. |
| Tolerance | Retries | Successive attempts to perform an action (e.g., requests) upon an error caused by transient faults. |
| | Fallbacks | Masking faults of read-only operations with default responses, e.g. using static or cached values. |
| | Health checks | Probing the status of a microservice, e.g., availability, connections state, circuit breakers. |
| | Replication | Multiple distributed replicas to improve system availability and scalability. |
| | Load balancing | Workload distribution among existing microservice replicas to improve availability and scalability. |
| | Horizontal auto-scaling | Dynamic adjust on the number of replicas to meet current workload. |
| Removal | Monitoring | Collecting, processing, aggregating, and displaying real-time quantitative data about a system, e.g., query counts and types, error counts and types, processing times, and server lifetime. |
| | Chaos engineering | Checking system dependability by raising fault hypothesis, taking possible inputs, running experiments in production, and automating the experiments. |
| | Canary deployments | Verifying microservices for faults and anomalies by progressively deploying new versions of the application in production, having the workload split between the stable and the new version. |
| | Automatic diagnosis | Automating root cause analysis by detecting, localizing, and testing hypothesis of faults. |
| | Formal methods | Modeling microservices in theoretical domain, where system correctness can be confirmed or refuted. |
| Forecast | Fault tree analysis | Offline analysis of faults by modeling the system structure to infer the causes of events from boolean operators. |
| | Failure prediction | Online machine learning techniques to predict faults based on the past events and collected metrics. |

**Table 3.** Common countermeasure techniques to handle faults and failures in microservices.

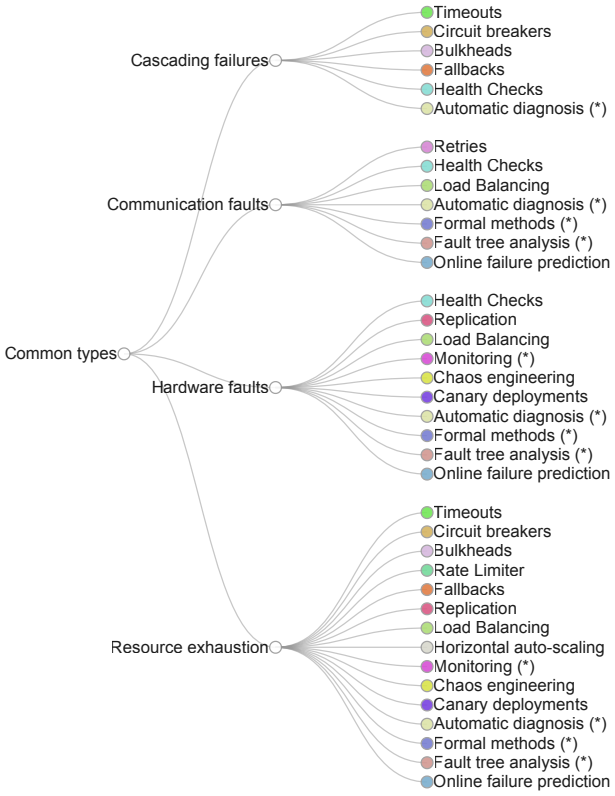


**Figure 1.** Common types of faults and failures and the corresponding countermeasure techniques. (*) Technique that handles indirectly the fault or failure.

failures and resources exhaustion, timeouts and rate limits can be set [Richardson, 2019].

**Fault tolerance.** It prevents faults to become failures through error detection and recovery [Avizienis *et al*., 2004]. Common implementation techniques for microservices are: successive service request retries [Brooker, 2019]; tolerating a fault by masking it through fallbacks [Basiri *et al*., 2016]; probing microservices to check whether they are healthy [Homer *et al*., 2014; Burns *et al*., 2016]; using multiple instances of the same microservices [Fowler, 2016]; balancing the workload among replicas [Newman, 2015]; scaling horizontally the resources capacity (e.g., replicas) to dynamically adapt to the current workload [Richardson, 2019; López and Spillner, 2017].

**Fault removal.** It aims to reduce the number of faults or its severity in both development and in production environments by applying techniques of verification, diagnosis, and fault correction [Avizienis *et al*., 2004]. Common implementation techniques for microservices are: making the system observable [Niedermaier *et al*., 2019] with quantitative data (logs, metrics, and traces) to monitor it [Beyer *et al*., 2016]; verifying system dependability with chaos engineering [Basiri *et al*., 2016; Heorhiadi *et al*., 2016; Blohowiak *et al*., 2016], i.e., by introducing and automating experiments in production; splitting workload between stable and new application version with canary deployments [McCaffrey, 2015; Schermann *et al*., 2016; Rajagopalan and Jamjoom, 2015]; finding root causes with automatic failure localization and diagnosis [Kim *et al*., 2013; Wu *et al*., 2020; łgorzata Steinder and Sethi, 2004]; and modeling formal methods [Newcombe *et al*., 2015] to identify faults, e.g., specification languages [Lamport, 2002], and model checkers [Yang *et al*., 2009; McCaffrey, 2015].

**Fault forecasting.** It consists of assessing the system behavior in the presence of a fault but before it happens in production [Avizienis *et al*., 2004]. Such assessment can be qualitative, through identification and classification of fail-

ures, or quantitative, by calculating fault probability. The techniques may be applied offline or online (in runtime). Fault Tree Analysis (FTA) is an offline technique applied to microservices [Zang *et al*., 2019; Pai and Dugan, 2002; Lee *et al*., 1985]. Online fault forecasting is usually applied through machine learning models (e.g., bayesian, markovian, regression, classifiers, deep learning) to train and predict faults from data-sets of collected metrics [Grohmann *et al*., 2021; Salfner *et al*., 2010; Pitakrat *et al*., 2018; Samir and Pahl, 2019].

#### 3.4.4 Supporting technologies

Here, we present solutions to support dependability on microservices into four technology types: frameworks and libraries, platforms, third-party applications, and tools. Most of the existing technologies are of open-source projects made available by big players such as Google, Netflix, SoundCloud, and VMware. Refer to Table 9 in Appendix A, where we provide weblinks as references for each technology we mention in this section.

**Libraries and frameworks.** To prevent cascading failures, in 2012 Netflix realized the need for isolating integration points when their application was calling external services. In this context, `Hystrix` was implemented through a Java/JVM library, which provides protection against cascading failures while providing fault tolerance through circuit breakers, bulkheads, timeouts, and fallbacks. Since `Hystrix`, other libraries were built for different platforms and languages, e.g., `Steeltoe` for .NET, `Shopify Semian` for Ruby, and the `Hystrix`'s successor, `Resilience4j` for the JVM. Extending `Hystrix`, `Resilience4j` includes retries and rate limiters. With similar features in Java, one can mention `Eclipse MicroProfile`, and `Spring`. Frameworks such as `Spring Boot`, `ASP.NET Core`, `Micronaut` provide support for health checks, allowing microservices to expose metadata such as resources status information. `Spring Actuator` and `Prometheus` are libraries capable to collect metrics from multiple microservices and, through HTTP requests, make the metrics available to external tools to monitor the application.

**Platforms.** Cloud platforms such as Amazon AWS and Google Cloud Computing Platform offer products supporting fault countermeasure techniques (Section 3.4.3). `AWS ELB` provides load balancing and horizontal auto-scaling. Storage systems as `AWS S3`, `Google Storage`, `AWS RDS`, and `Google Cloud SQL` support data replication on multiple geographical zones to tolerate hardware faults and data loss. Since agnostic and reproducible environments are features desired to run microservices applications, containerization technologies such as `Docker` and `rkt` provide portable and consistent environment of application deployment. In this sense, container orchestration platforms are key enablers for microservices. `Apache Mesos` but mostly Google's `Kubernetes` is a widely used platform. `Kubernetes` is a very popular orchestrator which provides a uniform environment to run microservices on cloud platforms, allowing practitioners to implement load balancing, horizontal autoscaling, health checks, and replication.

**Third-party applications.** With the popularization of microservice architectural style, maintaining the libraries and frameworks in multiple languages and platforms may become unfeasible [Jamshidi *et al*., 2018]. For service meshes, `Linkerd` and `Istio` are independent applications running side-by-side to the services (i.e., sidecar pattern), capable to connect, secure, control, and observe microservices, besides enabling support for techniques such as circuit breaking, load balancing, retries, request timeouts and fault injection. With support of service meshes, microservice applications should not concern with the implementation of such techniques. Particularly, `Istio` is a popular solution which extends `Kubernetes` to enable microservices applications aware of network, providing traffic management, telemetry, and security services. To enable chaos engineering, Netflix's `Chaos Monkey` is an application used to verify the resiliency of the infrastructure layer by randomly terminating VMs. Since the advent of Netflix's `Simian Army` and `Chaos Platform ChAP` [Blohowiak *et al*., 2016], practitioners can inject faults and delays into microservices. One can mention other later applications, such as `Apache Chaos Toolkit`, VMware's `Mangle`, and `Litmus`. Commercial solutions were also proposed, such as `Gremlin`, which allows a wide range of fault injections into infrastructure and application layers. Crosscutting functionalities, e.g., logs, metrics collection, monitors, fault injection (chaos engineering), can be found in third-party applications, i.e., autonomous agents running out of the microservice application scope. `Graylog`, `Logstash` and `Loki` are able to collect, store and search for logs generated by microservices. `Prometheus`, `Metricbeat` and `Apache Skywalking` can collect metrics from the host operating system and the target application, exporting the metrics to external visualization tools. `Zipkin` and `Jaeger` can collect and exhibit microservices traces. `Grafana` and `Kibana` provide interfaces to visualize metrics, logs, traces, and manage alerts of abnormal conditions. Multi-functional commercial products are also available, such as `Dynatrace`, `Datadog`, `Sentry`, `Rollbar`, and `New Relic`.

**Tools.** General purpose dependability tools are mostly applied at the development time to check system functionalities and remove faults before going to production. Specification languages such as `TLA+` [Lamport, 2002] and model checkers such as `TLC` [Yu *et al*., 1999] allow documenting and verifying system design. Static analyzers such as `FindBugs`, `Coverity`, and `Infer` can catch critical errors, e.g., null pointer exceptions, leaks, concurrency and synchronization inconsistencies.

# 4 Methodology

In this section, we describe the methodology we adopted 1) to conduct the opinion survey with microservice practitioners, and 2) to review the literature.

## 4.1 Practitioners survey

We conducted the opinion survey in February 2024, collecting responses from industry practitioners to capture current perspectives. We describe details on the opinion survey methodology in the following.

#### 4.1.1 Study design

Since questionnaires are one of the main strategy of gathering relevant data [Wohlin *et al*., 2012], we conduct the opinion survey by preparing and disseminating a questionnaire to practitioners, i.e., technology information professionals who deal with microservices-based systems in their everyday work. We followed the opinion survey guidelines described in Molléri *et al*. [2016] to design the questionnaire. Our questionnaire consisted of characterization questions to outline the participants' profile, and technical questions related to dependability in microservices.

To focus on practitioners, we preserved the number of questions small enough to ensure more significant number of responses, while limiting the questionnaire to four sections only. The first section presents the context of the opinion survey and an informed consent form. The characterization questions are in the second section. In the third section, participants evaluated the types of faults and failures (Section 3.4.2) using a Likert scale for their responses. In the fourth section, participants were asked about countermeasure techniques (Section 3.4.3) they have been using, with option to inform others techniques. We also included open questions to understand which applications, products, libraries, and frameworks the participants have used to improve dependability in their applications. The list of all questions asked is shown in Table 4. The last three sections were only enabled to participants who agreed with the informed consent and confirmed that they held knowledge on microservices.

| ID | Sec. | Type | Question |
|---|---|---|---|
| Q1 | CH | Open | What is your country of residence? |
| Q2 | CH | Closed | What is the size of your company? |
| Q3 | CH | Closed | How long have you been using microservices? |
| Q4 | FP | Closed | How often have you experienced the fault patterns below? |
| Q5 | TC | Closed | Which techniques have you used to improve dependability of microservices applications? |
| Q6 | TC | Open | Which applications, products, libraries, and frameworks are you using to improve application dependability? |
| Q7 | TC | Closed | Which of the following inputs and tools have you used to diagnose failures on microservices? |
| Q8 | TC | Closed | In your opinion, what is the difficult level associated to debugging and diagnosing failures on microservices? |

Legend: CH (Characterization), FP (Fault Patterns), TC (Techniques and Countermeasures

**Table 4.** Questions asked to practitioners in the opinion survey.

#### 4.1.2 Pilot study

We conducted a pilot study to validate the questionnaire and ensure that it fits to our purpose. To this end, we invited four professionals from different companies and experience levels to respond the pilot questionnaire in order to evaluate it. A summary of the four pilot participants is the following: all of them were from Brazil; one of 1-year experience, working for a company of up to 100 employees; the other three were practitioners working for bigger companies, with up to 1K employees; two of them were of 3-year experience and the other one of 2-year. After analyzing their responses and feedback, we identify the need for minor adjustments, e.g., rewording to mitigate ambiguity, and adding new items to closed questions. Although these adjustments did not change the essence of the questionnaire, we decided to discard the pilot participants' responses to ensure the integrity of the final analysis.

#### 4.1.3 Participants selection

We disseminated the questionnaire through LinkedIn and Twitter, encouraging our network connections to share it and amplify its reach.

#### 4.1.4 Execution and data analysis

As mentioned earlier, we disseminated the questionnaire in February 2024. We collected a total of 46 responses and analyzed them using descriptive statistics.

### 4.2 Literature review

To provide a comprehensive understanding of academic contributions and provide an up-to-date overview of the state-of-the-art, we conducted a literature review starting from 2020. By examining recent developments in the field, we aim to bridge the gap between academia and industry, offering a well-grounded exploration that highlights how current research aligns with industry needs.

To meet the aim of this review, we took well-defined steps, which include: the definition of objectives and research questions, selection of search bases, selection criteria, and data extraction. Such steps are in accordance with typical literature review protocols [Petersen *et al*., 2015], meanwhile our review method does not strictly follow well-known systematic literature review (SLR) guidelines [Kitchenham and Charters, 2007]. In the following, we provide details of each step.

#### 4.2.1 Objective and research question

This literature review aims to identify and analyze the most recent approaches for detecting faults and ensuring reliability in microservices systems. Our research is particularly focused on the dimension of *countermeasure techniques*, i.e., typical methods to tackle faults and failures.

The research question we intend to answer through this literature review is:

> *What are the countermeasure techniques applied to make microservices more dependable?*

#### 4.2.2 Search bases and search string

For a straight away comparison of opinion survey responses with the contemporary research landscape, we limited our literature review to publications from 2020 onward. Such a comparison is crucial for identifying emerging trends and verifying whether recent developments in the field are reflected in professional opinions and practices.

| Criteria | Description |
|---|---|
| IC1 | Studies that address countermeasures techniques to improve the dependability of microservices. |
| IC2 | Studies that address tools and frameworks to diagnose failures in microservices. |
| EC1 | Studies that do not focus on microservices. |
| EC2 | Studies that do not address tools and frameworks to diagnose failures in microservices. |
| EC3 | Studies not written in English. |
| EC4 | Posters, short papers, surveys, and tutorials. |
| EC5 | Studies with no clear alignment among publication venue, the publication year, and the number of citations. |

**Table 5.** Criteria to include and exclude studies.

We selected ACM Digital Library[2] and Google Scholar[3] as primary search databases. Google Scholar particularly interesting once it allows extensive coverage that includes a wide spectrum of academic and gray literature sources. This decision aimed to maximize the breadth of our literature review.

The specific search string employed across these databases, designed to address our research question was:

```
(fault failure dependab* resilien*)
AND (microservice*)
```

#### 4.2.3 Studies selection

We conducted searches on two significant databases at different times in 2024 to capture the most relevant and recent literature. The initial search was carried out on Google Scholar in March, yielding 1,320 articles. We ordered these results by relevance and reviewed the abstracts, applying our inclusion (IC) and exclusion (EC) criteria described in Table 5. Given the vast number of results, we decided to halt the selection process after analyzing 50 consecutive works without finding any suitable for inclusion. The abstracts of 210 papers were reviewed, resulting in 47 being selected for a full analysis.

Subsequently, a more targeted search was conducted in the ACM Digital Library in June, which returned 99 results. This timing difference allowed us to incorporate the most recent publications, ensuring comprehensive coverage of the latest research. All 99 abstracts from the ACM search were reviewed using our inclusion and exclusion criteria, identifying 38 relevant papers. Among the selected documents, 6 were identified as duplicates between the two databases. After consolidating the results and removing duplicates, we selected 79 documents for complete analysis. During the complete reading, we excluded 39 documents that did not address the relevant themes for this study, finalizing the selection of 40 documents for detailed review.

#### 4.2.4 Data extraction

Since this review does not adhere to all the traditional steps outlined in the Systematic Literature Review (SLR) guidelines by [Kitchenham and Charters, 2007], it does not include bibliometric data extraction or scientometric analysis. Instead, to conduct the literature review, we systematically organized, categorized, and connected the extracted data to identify countermeasure techniques researched since 2020. The findings from this literature review are presented in Section 6.

[2]https://dl.acm.org

[3]https://scholar.google.com.br

## 5 Results of the practitioners survey

In this section, we present results of the opinion survey.

### 5.1 Participants' characterization

Table 6 presents the participants' characterization from the responses to the questions Q1, Q2, and Q3. Professionals from three continents answered the questions, with the majority based in Brazil, where there were 28 responses (60.1%), indicating a bias related to the authors' nationalities. Participants from India represented 8.7% of the responses, while Portugal, Germany, and the USA each contributed 4.3%. The remaining participants were from several other countries. Four participants mistakenly provided non-country names when answering question C1, so we classify their origin as "N/A". A total of 54.4% of the participants work for companies with more than 1,000 employees, and 28.3% of them are employed by companies with over 10,000 employees. Regarding experience, more than 89% of the participants have been working with microservices for more than three years, indicating that the responses primarily come from professionals with relevant experience in the field.

| Continent | % | Company size | % | Experience | % |
|---|---|---|---|---|---|
| Americas | 67.4 | 1 employee | 4.3 | 0–2 years | 10.8 |
| Europe | 13.0 | < 100 employees | 10.9 | 3–4 years | 32.7 |
| Asia | 10.9 | < 1000 employees | 30.4 | ≥5 years | 56.5 |
| N/A | 8.7 | < 10,000 employees | 26.1 | | |
| | | ≥ 10,000 employees | 28.3 | | |

**Table 6.** Characterization of the participants. 46 in total.

### 5.2 Experienced fault patterns

For the question Q4, participants informed the frequency they perceive fault and failures, ranging from 0 (never) to 4 (very often). The heatmap in Fig. 2 illustrates how often practitioners experienced various fault patterns. While all the faults were experienced, the ones that most frequently occurred (i.e., often and very often) were the high memory usage and the high CPU usage, with 50% and 43.5% of the participants, respectively. Between 37% and 28% of the participants experienced often and very often TCP faults (37%), application layer faults (35%), configuration faults (35%), thread/connection pool exhaustion (28.3%), and cascading failures (28.3%). On the other hand, 80% and 72% of practitioners answered that they never experienced or experienced on rare occasions the hardware faults and disk space exhaustion, respectively.

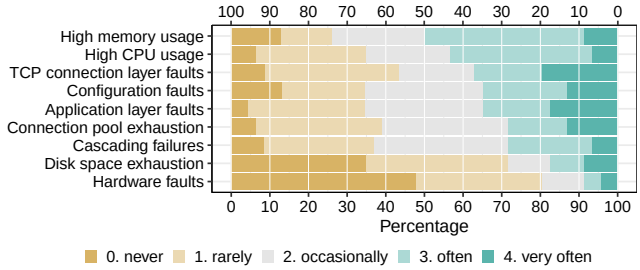


**Figure 2.** (Q4) General percentages of how often practitioners have experienced fault patterns.

## 5.3 Adopted countermeasure techniques

We list the techniques to improve microservice dependability in Q5 question, allowing participants to indicate which ones they have used. As shown in Fig. 3, the most commonly used techniques were Load Balancing, Retries, Circuit Breakers, Health Checks, and Monitoring, cited by 8/10 participants. Horizontal Auto-scaling and Timeouts were also popular techniques, used by 7/10 participants. Fallbacks and Replication were employed by 50% of the participants, while Canary Deployments were used by 41%. More advanced techniques, such as Bulkheads, Chaos Engineering, Automatic Failure Diagnosis, Fault Tree Analysis, and Formal Methods, were the least used, adopted by fewer than 15% of the participants.

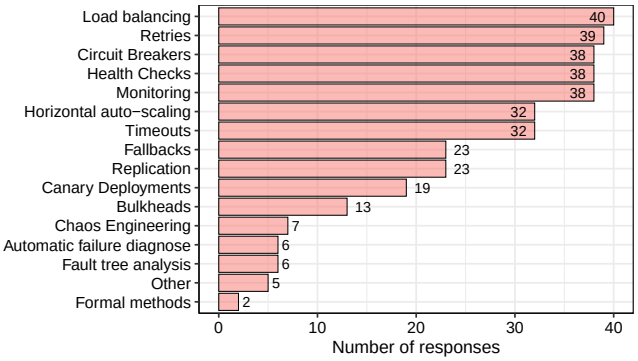


**Figure 3.** (Q5) Total numbers of responses from 46 participants in 2024 on which countermeasures techniques they have used to improve dependability of microservices.

## 5.4 Supporting technologies for dependability

Participants were asked to list the applications, products, libraries, and frameworks they use to improve the microservices dependability (open question Q6). Fig. 4 presents a word cloud of the responses. We observed the prevalence of `Kubernetes` as the main technology to improve microservices dependability as more than 50% of the participants mentioned it. `Kubernetes` was followed by `Istio`, `Spring`, and `Hytrix`, with 19.6%, 8.7%, and 6.6% of mentions among the participants, respectively.

## 5.5 Diagnosing faults and failures

Question Q7 sought to identify which data sources and tools are used by the professionals to diagnose microservice faults and failures. While 98% participants made used of logs, at least 83% of them also used metrics and traces. Third-party vendor tools for diagnosing were mentioned by 48% of the participants, while 24% of them made use of tools developed by their own companies.

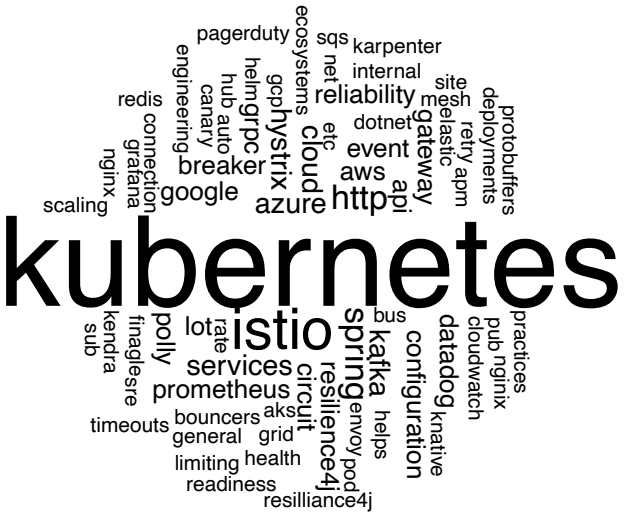


**Figure 4.** Applications, products, libraries, and frameworks to improve the microservices dependability used by the participants.

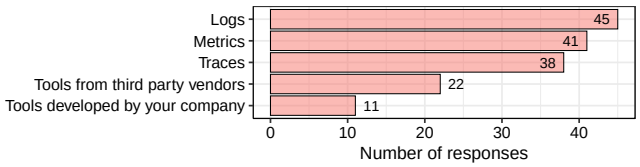


**Figure 5.** (Q7) Total numbers of responses from 46 participants on which inputs and tools they have used to diagnose failures on microservices.

From question Q8, we caught participants' perception on difficulties with debugging and diagnosing faults and failures in microservices applications. Despite having inputs and tools, most of the participants (63%) mentioned a high degree of difficulty (i.e., difficult and very difficult) in debugging and diagnosing faults. The remaining participants answered that such a task was neutral (28.3%) and simple (8.7%), while no one classified it as very simple.

# 6 Results of the literature review

This literature review provides a comprehensive analysis of recent research published from 2020 onwards, specifically addressing the research question discussed in Section 4.2.1. A trend identified is the predominance of automatic diagnosis and fault injection techniques as countermeasures techniques to enhance the dependability of microservices-based architectures. Additionally, we have identified considerable research contributions in areas such as failure prediction, monitoring, testing, and circuit breakers. Table 7 summarizes the key techniques identified in our literature review, categorized by their application and primary studies. These topics are explored in depth in the following subsections.

| Goal | Countermeasure | Reference |
|---|---|---|
| Prevention | Circuit breakers | Sedghpour *et al*. [2021]; Meiklejohn *et al*. [2022] |
| Removal | Canary Deployments | Gazzola *et al*. [2020] |
| | Monitoring | Scrocca *et al*. [2020]; Belkhiri *et al*. [2023] |
| | Chaos engineering | Camilli *et al*. [2022]; Poltronieri *et al*. [2021, 2022] |
| | Chaos engineering (Fault injection) | Meiklejohn *et al*. [2021]; Assad *et al*. [2024]; Wu *et al*. [2023]; Simonsson *et al*. [2021]; Zhang *et al*. [2021] |
| | Automatic diagnosis (RCA) | Gu *et al*. [2023]; Yang *et al*. [2023]; Yu *et al*. [2023]; Bi *et al*. [2022]; Zheng *et al*. [2024]; Harsh *et al*. [2023]; Gan *et al*. [2021]; Zhu *et al*. [2024] |
| | Automatic diagnosis (Anomaly detection) | Chen *et al*. [2022]; Bento *et al*. [2021]; Guo *et al*. [2020]; Xie *et al*. [2023b,c]; Zhao *et al*. [2023]; Huang *et al*. [2022]; Hrusto *et al*. [2022]; Xie *et al*. [2023a] |
| | Automatic diagnosis (Autonomous correction) | Sarda *et al*. [2023]; Lee *et al*. [2023]; Ma *et al*. [2020]; Yang *et al*. [2021] |
| Forecast | Failure prediction | Rouf *et al*. [2024] |
| | Simulation | Frank *et al*. [2022] |

**Table 7.** Summary of countermeasure techniques for dependability in microservices found in the literature review.

## 6.1 Automatic diagnosis

In recent literature on automatic diagnosis in microservices systems, we observed several studies that deal with detecting and diagnosing anomalies. We classify such approaches into three main categories, each exploring different techniques and models: approaches that focus exclusively on root cause analysis, those that focus on anomaly detection, and those that focus on integrating anomaly detection and automated diagnosis.

### 6.1.1 Root cause analysis

Root Cause Analysis (RCA) in microservices systems has evolved in recent years, adapting to incorporate multimodal data (logs, metrics, and traces) and applying advanced machine learning and artificial intelligence techniques.

TrinityRCL [Gu *et al*., 2023] and `Hi-RCA` [Yang *et al*., 2023] are RCA frameworks which employ combination of metrics, logs, and traces. The former builds causal graphs that represent the relationships between entities associated with anomalies. On the other hand, the latter adopts a two-phase approach, starting with Kalman filtering to quantify abnormalities, followed by correlation analysis to identify anomalous metrics and root causes down to the code level. Similarly, Nezha [Yu *et al*., 2023] also uses multimodal data to locate root causes at granular levels, such as code regions and types of resources. This technique transforms heterogeneous multimodal data into a homogeneous representation of events, which facilitates the extraction of event patterns by constructing and mining event graphs. Mulan [Zheng *et al*., 2024] unifies causal analysis in multimodal systems, transforming logs into temporal data and using contrastive learning to highlight causal relationships.

Murphy [Harsh *et al*., 2023] is designed to diagnose performance issues within enterprise environments, focusing on uncovering hidden faults and inferring causal relationships among entities and specific metrics that contribute to observed problems. In contrast, Sage [Gan *et al*., 2021], an RCA system for cloud microservices, leverages unsupervised machine learning techniques to pinpoint the causes of unpredictable performance. This system streamlines the diagnostic process by eliminating the need for manual data labeling, thus enhancing problem identification in dynamic settings. Additionally, MicroIRC [Zhu *et al*., 2024] targets root cause identification at the instance level within microservices, utilizing metric-based data to provide precise diagnostic insights.

VecroSim [Bi *et al*., 2022] is a tool that performs metric-oriented fault simulations, generating datasets with abnormal performance metrics on demand that can be used to assess different techniques of root cause analysis.

### 6.1.2 Anomaly detection

In general, most approaches to anomaly detection in microservices systems employ machine learning techniques, using monitoring data that includes performance metrics such as CPU usage, memory, and latency. These data are often treated as multivariate time series, allowing for a detailed analysis of system behavior over time.

Chen *et al*. [2022] introduce DAM (Deep Attentive Anomaly Detection with Multimodal Data), which uses multimodal data to detect when and where anomalies occur in microservices systems. This method employs a predictive model that uses LSTM (Long Short-Term Memory) and attention mechanisms to forecast future values and identify deviations from these forecasts as anomalies. Additionally, Bento *et al*. [2021] developed the OTP (OpenTracing Processor) tool, which extracts and analyzes metrics from data in the OpenTracing format. Using machine learning models, OTP analyzes this data to identify operational patterns that deviate from the norm, pointing out services that may be experiencing issues.

Guo *et al*. [2020] propose GMTA (Graph-based Microservice Trace Analysis) to process real-time generated traces, abstracting them into different paths and grouping them into business flows. This method combines a graph database with a real-time analytics database for more effective analysis. On the other hand, GTrace [Xie *et al*., 2023b] categorizes traces into groups based on their shared substructures, such as entire trees or subtree structures. Additionally, Xie *et al*. [2023c] in-

troduces TraceVAE, a dual-variable graph variational autoencoder that models structural and temporal characteristics, addressing the inversion of negative log likelihood (NLL) with specific techniques to minimize entropy differences.

AnoFusion [Zhao *et al*., 2023] presents an approach that leverages unsupervised multimodal data for failure detection in microservices systems. Such an approach starts by serializing multimodal data into time series, then, AnoFusion constructs heterogeneous graphs that effectively capture the intricate connections and interactions among datasets. This approach enables the identification of critical patterns and trends that can signify existing anomalies and also predict potential failures. SLA-VAE [Huang *et al*., 2022] is an approach tailored for industrial environments such as cloud servers or microservices. This framework comprises three main components: anomaly feature extraction that quantifies deviations from historical data, a semi-supervised variational autoencoder that enhances robustness and reduces susceptibility to anomalous inputs, and active learning that updates the anomaly detection model online using a small set of uncertain samples.

Deep learning methods for anomaly detection in multivariate time series are exploited in Hrusto *et al*. [2022], integrating a filter into a DevOps environment and utilizing continuous feedback from the development team. Finally, ImpactTracer [Xie *et al*., 2023a] focuses on modeling the propagation of failures in microservices systems. Using the Isolation Forest model, based on decision trees, ImpactTracer detects anomalies using business golden indicators, such as response and success rates. This system constructs an impact graph that maps all possible paths of failure propagation, allowing for the evaluation of failure probabilities from one node to another, thus improving accuracy in failure prevention and correction.

#### 6.1.3 Integrating anomaly detection and automated diagnosis

This category encompasses approaches in which intelligent systems are capable of not only identifying problems but also suggesting or executing autonomous corrections.

ADARMA platform, proposed by Sarda *et al*. [2023], uses Large Language Models (LLMs) for complete automation, from anomaly detection to failure remediation. ADARMA collects system logs and metrics, identifying anomalous behaviors and performing RCA through probabilistic reasoning and historical data. The integration of prompt engineering allows for the generation and execution of automatic remediation code, efficiently addressing faults.

Similarly, the Eadro framework Lee *et al*. [2023] and AutoMAP [Ma *et al*., 2020] also integrate detection and diagnosis, automatically facilitating problem resolution and minimizing manual intervention. Eadro is an end-to-end framework that treats anomaly detection and root cause localization as interdependent tasks, using multitask learning for a more integrated approach. In turn, AutoMAP automatically creates an anomaly topology with no need for prior knowledge of the system architecture, using data collected from microservice requests and container/host states.

Furthermore, Yang *et al*. [2021] introduces AID (Approach to Intensity of Dependency), a methodology developed to diagnose failures and enhance recovery in cloud-based systems, considering the intensity of dependencies between microservices. According to the authors, AID improves efficiency when identifying the root causes of failures and optimizes the systems' subsequent recovery.

### 6.2 Chaos engineering

In Chaos Engineering, many studies focus on fault injection techniques. Differently, Poltronieri *et al*. [2021] and Poltronieri *et al*. [2022] introduce the ChaosTwin framework, which applies the principles of chaos engineering using digital twins, allowing virtual representations of real systems. Such a framework allows for testing resilience of IT systems by simulating failures in their digital twin, assessing configuration and failure management strategies from a business impact perspective.

#### 6.2.1 Fault injection

Various techniques have been developed in the field of fault injection in microservices. Approaches such as SFIT (Service-Level Fault Injection Testing) [Meiklejohn *et al*., 2021] and Filibuster [Meiklejohn *et al*., 2021; Assad *et al*., 2024] integrate fault injection early in the development cycle to detect and mitigate faults in advance. SFIT integrates fault injection to introduce specific fault scenarios from the start of development and combines static analysis and concolic execution with a dynamic reduction algorithm, aiming to reduce the number of test cases generated without compromising fault coverage. The Filibuster tool implements SFIT, facilitating early detection of problems, while eliminating the need for chaos experiments in production environments. Additionally, Assad *et al*. [2024] extends Filibuster([Meiklejohn *et al*., 2021], focusing on fault injection in database clients within microservices applications, supporting both SQL and NoSQL systems, and integrating this functionality during the development phase.

In Wu *et al*. [2023] is introduced the concept of "Fault-Tolerance Bottleneck", which aims to identify a minimal set of faults whose injection can interrupt all possible execution paths in a microservices system. Based on this concept, the authors proposed FBFI approach (Fault-tolerance Bottleneck driven Fault Injection), which allows to validate correction of each deployed component, without the need for complete prior knowledge of the system structure.

CHAOSORCA [Simonsson *et al*., 2021] is designed to inject faults into system calls in containerized applications. CHAOSORCA focuses on assessing self-protection capacity of microservices applications upon errors in system calls. Operating under workload conditions similar to those found in production environments, CHAOSORCA has no need for application instrumentation. 3MileBeach [Zhang *et al*., 2021] is a platform that combines fault tracking and injection. It uses Temporal Fault Injection (TFI) technique, which allows controlling the flow of messages between services with specific temporal requirements. TFI focuses on the temporal precision of interactions between services, allowing develop-

ers and testers to simulate faults at very specific points into interaction flows.

### 6.3 Failure prediction

Although some anomaly detection techniques can also be used for failure prediction [Zhao *et al*., 2023; Huang *et al*., 2022], we found InstantOps [Rouf *et al*., 2024] as the only one which is specifically focuses on failure prediction. InstantOps employs an approach that combines multimodal data to anticipate failures and conduct RCA. This methodology uses machine learning techniques to integrate and analyze spatial and temporal data from various sources. Failure prediction is achieved by identifying patterns that indicate imminent failures based on the dynamic interaction between microservices components represented in a dependency graph. After predicting failures, the system uses node scores generated by the model to specifically identify which components or services might be causing problem.

### 6.4 Monitoring and testing

Some techniques such as MIPaRT [Camilli *et al*., 2022], ExVivoMicroTest [Gazzola *et al*., 2020], and Kaiju [Scrocca *et al*., 2020], present approaches to monitoring and testing microservices. MIPaRT is an ex-vivo framework that integrates into the DevOps cycle, enabling continuous testing and monitoring. This framework automates the generation and execution of performance and reliability tests, collects and analyzes monitoring data, and offers integrated visualization of results. Concurrently, ExVivoMicroTest enhances regression testing by collecting service interactions directly from the production environment and converting them into test cases that reflect real usage, thus addressing the gap between idealized test scenarios and actual operational conditions. Additionally, Kaiju is focused on observability in microservices, facilitating the integration and analysis of metrics, logs, and trace data through an event-based model. This approach provides a unified and detailed view of system behavior.

Extending these capabilities, Belkhiri *et al*. [2023] propose an approach for annotating application traces with kernel-level data without the need to modify the application, the trace system, or the operating system. This technique focuses on diagnosing causes of longer latencies by integrating kernel information into traces from common distributed tracing tools.

### 6.5 Circuit breakers

Sedghpour *et al*. [2021] introduces an adaptive circuit breaker mechanism that prevents overload and ensures efficient response time. Such a mechanism dynamically adapts to changes in workload, server capacity, and service complexity. By utilizing control theory principles, it adjusts the activation thresholds of the circuit breaker so as to respond more accurately to system variable conditions. Meanwhile, Meiklejohn *et al*. [2022] offer a taxonomy for circuit breakers and propose the design of two new ones. The new circuit breakers can handle more effectively abstraction and scope within application code, dynamically adapting to changes in operational conditions in real-time. Unlike traditional models that often apply generic and broad interruptions, these new designs for circuit breakers allow for more granular control and transparent integration, facilitating continuous application maintenance and development. These approaches represent a significant advancement in how microservices systems can remain resilient and efficient to face potential failures and operational variations.

### 6.6 Others

Other approaches explore issues concerning with antifragility [Bangui *et al*., 2022], redundancy [O'Neill and Soh, 2023; O'Neill and Soh, 2022], and latency prediction [Tam *et al*., 2023] in microservices. MiSim [Frank *et al*., 2022] stands out for its wide applicability in many countermeasure techniques. MiSim [Frank *et al*., 2022] is a simulator designed for assessing resilience of microservice-based architectures. It facilitates a holistic assessment of resilience, allowing for the configuration and execution of experiments that simulate adverse conditions to observe how architectures respond to such situations. To this end, MiSim integrates and simulates a variety of resilience mechanisms, including circuit breakers, retries, load balancers, and autoscalers.

Bangui *et al*. [2022] propose a conceptual framework based on antifragility to allow microservices to learn and strengthen from adversities. Such a framework aims to protect critical infrastructures (CIs) against unexpected events, encouraging exposure to uncertain conditions to develop active resilience, leveraging from the system's own disorders.

The resilience of cloud systems based on microservices is further exploited by O'Neill and Soh [2022], who introduced an approach based on Task-Based Reliability (TBR). The technique optimizes redundancy of microservice containers according to specific user tasks, taking into account the trade-off between the marginal cost of added redundancy and the expected cost of failure. Building on this, O'Neill and Soh [2023] later introduced an orchestration algorithm that utilizes the TBR model to manage redundancy efficiently. The model dynamically adjusts to spot market changes, aiming to balance operational costs with resilience requirements.

Tam *et al*. [2023] make use of graph neural networks to predict end-to-end latency in microservices applications. This framework enhances accuracy of latency predictions and provides a solid foundation for proactive resource scaling, improving reliability and efficiency in dynamic environments.

## 7 Discussion

Here, connecting the results of both the opinion survey and literature review, we discuss the main findings, challenges, and opportunities identified.

### 7.1 Key findings from industry practices

From the opinion survey results, we highlight key findings regarding the adoption of microservices and the industry's

approach to dependability as follows.

#### 7.1.1 Microservice adoption

By analyzing the responses from practitioners in the opinion survey, we observe widespread adoption of microservices. Companies, regardless of size, are adopting the microservices architectural style as a way to give autonomy to their teams and speed up development. As more companies adopt it, opportunities to gain practical experience expand, leading to greater proficiency within the community. Most participants (56.5%) reported having 5 or more years of experience, and this professional experience combined with adoption by companies creates a positive cycle.

#### 7.1.2 Diverse patterns of faults and failures

The fault and failure occurrence patterns faced by practitioners were as follows.

**Rare patterns**. We observed that hardware and disk exhaustion faults are infrequent in production environments, thanks to the fault tolerance mechanisms in place, such as replication, health checks, and horizontal auto-scaling. The adoption of cloud providers and container orchestrators also contribute to this scenario, as they offer built-in features and services to manage hardware faults effectively.

**Balanced patterns**. In examining fault patterns among practitioners, we found moderate stability but noted significant areas of concern. TCP connection layer faults stand out, with 43% rarely or never encountering them, while 37% experience them often or very often. Configuration and application-level faults share a similar pattern, with 35% for never-rarely and 35% for often-very-often. Connection pooling issues are infrequent for 39% of practitioners but common for 28%. Lastly, cascading failures are rarely or never observed by 37% yet frequently for 28%. Despite this balance, these faults and failures were a notable concern for a significant portion of practitioners, highlighting areas where robust fault-tolerance mechanisms are essential.

**Frequent patterns**. Despite the continuous increase in resources and computing power, failures due to resource exhaustion are still common in microservices applications, mainly related to memory and CPU usage, with 50% and 43% of practitioners, respectively, experienced them often and very often. Applications are subjected to poorly development, which might lead to memory leaks. Today's applications are more data-intensive (more data is collected and transformed), which demands resources.

#### 7.1.3 Most and least used countermeasure techniques

We observed that the preference of most practitioners are similar across the techniques. We highlight the following:

**Popular techniques are for fault tolerance, prevention, and removal**. Nowadays, there is a rich ecosystem of tools, libraries, and platforms equipped with implementation of techniques to prevent and tolerate faults (see Figure 6 in Appendix A). Driven by the goals in Table 3, we observed that the most commonly used techniques (i.e., those applied by at least 50% of practitioners) were primarily focused on providing fault tolerance, including Load Balancing (87%), Retries (85%), Health Checks (83%), and Fallbacks (50%). Secondly, to achieve fault prevention, practitioners reported using Circuit Breakers (83%), Horizontal Autoscaling (70%), Timeouts (70%), and Replication (50%). Finally, for fault removal, practitioners predominantly applied Monitoring (83%).

**Fault removal and forecasting remain underused**. With the exception of monitoring, techniques for these goals were not frequently mentioned by practitioners. Few professionals reported using related techniques such as formal methods, fault tree analysis, chaos engineering, and automatic diagnosis. This can be explained by the lack of standardized tools and the complexity of these countermeasures.

#### 7.1.4 Supporting technologies

As for the technologies to support dependability, we found:

**Container-based sidecar facility**. We observed a trend to adopt platforms based on containers and sidecar patterns, e.g. `Kubernetes`, and `Istio`. Firstly, as a way to transparently add a number of countermeasures to the whole system without requiring language-specific libraries. Secondly, to be independent of underlying cloud platforms.

**`Kubernetes` at the top of technologies.** `Kubernetes` has become the *de facto* choice for orchestrating applications, being mentioned by 50% of practitioners. Such a strong popularity is justified by a highly active open-source project, with several important patterns implemented in it, cloud independence support (multi-cloud possibility), and high application portability (containers).

**A number of other technologies**. While container orchestration, service meshes, and libraries are spotted with technologies of reference (i.e., `Kubernetes`, `Istio`, and `Hystrix`, respectively), there are no best choice technologies for most applied techniques such as load balance and monitoring, and powerful practices such as chaos engineering. Various technologies were mentioned by practitioners.

#### 7.1.5 Complexity of diagnosing faults

Logs, metrics, and traces were identified as the primary sources of data for diagnosing faults and failures, with 98%, 89%, and 83% of participants, respectively, relying on them. Additionally, 48% of practitioners reported using third-party vendor tools, while 23% relied on tools developed by their companies. Despite having access to these data sources and tools, diagnosing faults and failures remains a complex task for microservices developers, according to more than 60% of survey participants. Various aspects contribute to a highly complex environment, e.g., diverse patterns of faults, failures propagate across microservice instances, intricate interactions among distributed services and their dependencies, and large amounts of input data are generated by monitoring tools. As a result, even experienced practitioners – 56% of whom have over 5 years of experience – find it challenging to identify the root cause of faults and failures.

## 7.2 Industry practices and academic research

Upon analyzing the two sides, i.e., current industry practices and academic efforts between 2020 and 2024, the connections between them we discuss in the following.

**Fault removal, an industry gap yet in papers**. As highlighted in Section 6, the predominant focus for the academic community has been developing fault removal techniques. Particularly, automatic diagnosis has been gathering much attention. Nevertheless, as outlined in Section 5, according to the practitioners survey results, fault removal represents the less applied means to enable dependability. A possible reason for the underutilization is that the involved technologies might yet need to achieve higher maturity levels for a broad application in industrial settings. On the other hand, extensive research on this theme suggests that academia is aware of the industry gaps, actively working to tailor solutions aligned with the practical needs of industry. Furthermore, another potential reason for a slow industrial adoption is that the industry has not sufficiently disseminated or embraced the state-of-the-art advancements made in academia

**Towards autonomous failure diagnosis, while logs are manually inspected in production.** From the practitioners survey, we observed that logs are by far the main data source used by human operators to diagnose issues in production. Although metrics and traces are also commonly used, the fault diagnosis process in industrial contexts is still prone to be manually accomplished. In response to this limitation, researchers have been proposing multimodal automated support mechanisms, integrating advanced machine learning techniques to predict patterns and anticipate failures (e.g. Gu *et al*. [2023]; Yang *et al*. [2023]; Yu *et al*. [2023]; Zheng *et al*. [2024]). Such investigations demonstrate improvements in diagnostic accuracy and efficiency, showing their potential for reducing downtime and increasing productivity in industrial environments.

**Well-established technologies also have room for improvements.** Despite their proven usefulness in industrial applications, there is a lack of studies focused on fault prevention and tolerance techniques. This is somehow contradictory, especially considering the critical role such techniques play in ensuring reliability to microservices. One of the factors that may contribute to that scenario is the widespread adoption of well-established technologies such as `Kubernetes`, which already offer effective solutions to fault prevention and tolerance. However, our literature review highlights ongoing efforts to refine such techniques. Recent studies exploit different designs of circuit breakers aiming to optimize time in *open* state, suggesting that countermeasures techniques even implemented in well-established technologies can be improved to provide performance gains in latency, availability, and reliability.

**Emerging contexts off the practitioners' radar.** From the literature review, we noticed that several studies address microservices in specific contexts with application scenarios, mostly in Edge Computing, Cloud Computing, and critical infrastructures. When analyzing responses to the open question Q6 (on application, products, libraries, and frameworks), we didn't observe trends in applications. In this sense, the increasing amount of research with specific contexts and application scenarios suggests that problems and challenges addressed in academia might not yet be fully recognized or addressed in current industry practices.

## 7.3 Research challenges and opportunities

From the results of the opinion survey and literature review we conducted, we discuss the main challenges and opportunities we found in the following.

**Fault removal and forecast.** From the means to improve microservices dependability, we observed that fault removal and fault forecasting are the least applied. Techniques like fault-tree analysis, formal methods, canary deployments and chaos engineering are complex and have a steep learning curve. In this sense, we see an opportunity to have a better tooling around them, reducing the complexity of its application by abstracting details. Recent advancements from machine learning, such as the use of LLMs, may also reduce the adoption effort and increase the accuracy of automatic failure diagnose methods.

**Fault diagnosis.** We observe that fault diagnosis remains an open challenge regarding microservices dependability. Such a diagnosis ultimately depends somehow on human inspection, while being complex and requiring substantial knowledge of the target system. Even from existing observability tooling, fault diagnosis is exhausting due to the amount of data collected to be analyzed by engineers. Much effort and significant amount of time may be spent 1) to identify, analyze, and interpret the events that led to a failure, and 2) to create relevant documentation for continuous system improvement. In this context, we see research and development opportunities to achieve better methods and tooling to diagnosis faults.

**Open experimental platforms.** The lack of production-like environments makes difficult to evaluate new methods and techniques to improve dependability, pushing researchers to try to reproduce systems developed by the industry. Meanwhile, companies in general are not willing to grant access for their implementation details to competitors. In this context, as an enabler to research new techniques, building open experimental platforms and improving tooling for better synthetic microservices environments able to reproduce real production scenarios are interesting opportunities.

# 8 Threat to validity

Here, we outline potential threats to the validity of our study, which incorporates both opinion survey and literature review methodologies.

## 8.1 Opinion survey threats

- A risk of existing participants who misunderstand questions. To mitigate such a risk, we conducted the pilot study with a constrained number of participants, who gave us feedback on the questions.
- Undesirable audience (i.e., people who do not hold experience or knowledge in microservices) can respond

the questionnaire as it was publicly available. To prevent this, we disseminated the questionnaire exclusively to our network contacts. In addition, we emphasized in the questionnaire description that the target audience are developers, software architects, testers, and managers who deal with microservice-based systems. Finally, the questionnaire was only enabled for participants who confirmed, explicitly through a question in the questionnaire itself, that they had experienced in the topic.
- Survey respondents who do not represent completely the target population (i.e., microservice practitioners). To reach a broader and more diverse population, we disseminate the questionnaire through our international contacts who have significant influence in the field.

## 8.2 Literature review threats

- We provided the details of our methodology to allow the reproduction of the survey. However, the results obtained by performing the searches are dynamic in time, depending on the criteria used by the search engines to rank the studies.
- The literature selection process was guided by a search string. Some studies may use a different vocabulary than the words we used.
- Another related issue concerns subjectivity during the selection of studies. The authors used their background knowledge and research experience to decide on the relevance of the studies.
- Moreover, as microservices are an attractive and evolving topic, we might miss recent studies due to the timeline of the development of this survey.

# 9 Conclusions

As microservices are increasingly adopted, modern software systems are transitioning from monolithic to distributed and granular architectures with independent and executable artifacts. As an evolution of the client-server model [Salah *et al*., 2016], microservices offer autonomy and faster development for developer teams to build products more quickly while minimizing dependencies and conflicts. However, distributing functionalities into microservices increases system complexity, as each microservice can introduce potential points of failure.

In this paper, we shed light on recent industry practices to make microservice-based systems more dependable. Through an opinion survey with practitioners, we identified how they address faults, countermeasure techniques, applied technologies, and the challenges of diagnosing issues in production environments. While there is a strong preference for `Kubernetes` as a container orchestrator, we observed increased adoption of techniques such as load balancing, retries, circuit breakers, health checks, monitoring, and horizontal auto-scaling. However, fault removal and forecasting techniques receive limited attention. From a literature review covering the period between 2020 and 2024, we identified related topics addressed by academia. Using this two-fold methodology, we discuss key findings from current industry practices, correlate current research efforts with industry gaps, and finally, identify challenges and opportunities. In Table 8, we summarize the answers to the research questions of this work.

In future work, we plan to include more open-ended questions to capture diverse professional perspectives, enhancing the connection between their experiences and our conclusions. We also aim to conduct interviews for a qualitative analysis, providing deeper insights into the challenges professionals face regarding microservice dependability.

This research involved the collection of audio from beehives, exploring the potential of MFCCs and Mel spectrograms to describe colony strength. The main objective was to provide relevant information to beekeepers, assisting them, for example, in selecting hives for honey extraction. During development, we identified that to provide more practical guidance to beekeepers, classify the colony strength as strong or weak, simplify the decision-making process, and intervene when necessary. Additionally, audio capture in the apiaries allowed the construction of a labeled and public database, filling a gap in the literature.

The method proposed in this research achieved high accuracy using a descriptor with 40 MFCCs, overperforming CNN-based descriptors. Deeper models (VGG and ResNet) captured more useful features than the light models (MobileNet and YOLO), achieving better classification accuracy. These findings contribute to the specific understanding of colony strength classification and provide broader insights into the applicability of deep learning techniques in complex acoustic contexts. They enrich the ongoing discourse for future research and advancements in the field. The result suggests that a compact descriptor effectively identifies colony strength, offering a practical advantage: descriptors based on a single feature reduce the number of necessary calculations and extraction time. Moreover, they are more suitable for implementation on devices with limited computational power, often found in beehive monitoring environments in the apiaries. This consideration suggests that the proposed method is feasible for practical use in real-world conditions.

The nature of the field of beekeeping is complex. Environmental conditions and various factors can lead to overlaps in hive characteristics. Addressing this challenge is valuable, as it mirrors the real inherent complexity of hive monitoring. Additionally, it is crucial to emphasize the difficulty of collecting new samples, as it requires appropriate conditions in the hives and underscores the commitment to obtaining high-quality data.

For future research, we will investigate how noise filtering impacts preprocessing to improve classification performance and extrapolate the descriptor to other scenarios of interest in the beekeeping chain, such as identifying the presence or absence of the queen in the hive, detecting invaders, monitoring hive temperature, among other applications. Additionally, we evaluate the computational cost of different classifiers, including Markov chains, to embed the classification model and seek a representation of the colony strength with intuitive numerical values for beekeepers.

Another future research direction could be to investigate

| | |
|---|---|
| RQ1: | ***How is the industry dealing with dependability on microservices, in terms of experienced faults and failures, countermeasures applied (techniques and technologies)?***<br>The most frequent faults (§5.2) include high memory and CPU usage. The most common countermeasures (§5.3) are adopted for fault tolerance (Load Balancing, Retries, and Health Checks); for fault prevention (Circuit Breakers, Horizontal Auto-scaling, and Timeouts), and for fault removal (Monitoring). The main supporting technology (§5.4) used is `Kubernetes`. For fault diagnosis (§5.5), input data (logs, metrics, and traces) are widely employed, while most participants report significant difficulties in debugging and diagnosing faults, highlighting persistent challenges. |
| RQ2: | ***What are the main gaps and difficulties in terms of making microservices more dependable?***<br>The main challenges to increasing the dependability of microservices include mainly faults and failures of frequent and balanced pattern of occurrence (§7.1.2), coupled with the complexity of fault diagnosis (§7.1.5). Advanced techniques for fault removal and prediction are underused due to the lack of standardized tools and implementation complexity (§7.1.3). |
| RQ3: | ***What related topics have been researched by academia in relation to the opinion survey? How much are they correlated with industry gaps?***<br>Academia has been focused on automatic diagnosis, fault injection and prediction, monitoring, testing, and circuit breakers (§7.2). Although these topics have been exploited, the efforts have not yet translated into effective adoption by the industry, which continues to face challenges in fault diagnosis (§7.3). |

**Table 8.** Summary of Answers to Research Questions.

how MFCCs can be integrated as input to CNNs and evaluate the performance of these models in comparison with currently employed methods, bearing in mind that in the current work, CNNs were used only for feature extraction and not for classification. This approach would not only broaden the scope of input feature analysis but also offer valuable insights into the effectiveness of MFCCs in deep learning contexts for the task at hand.

We also aim to expand the applicability of the methodology developed in this study by increasing the number of samples, covering a variety of scenarios in beekeeping. A natural extension would be to explore the system's ability to identify the presence or absence of the queen in hives. Additionally, we consider integrating information on hive temperatures, a critical variable for bee health and productivity, investigating how the methodology adapts to apiaries in different regions and whether it will provide valuable insights for beekeepers and researchers. We also intend to explore the optimization of neural network architecture and model parameters to enhance the system's accuracy and efficiency further. Finally, we will develop robust models capable of extrapolating to various scenarios within hives, providing a versatile and valuable tool for monitoring and effectively managing bee colonies.

This study has consolidated the application of machine learning predictive models as a valuable tool for improving observability in complex IT systems. The microservices-based architecture proved to be the right selection, with significant benefits in terms of scalability and maintenance. The GradientBoostingRegressor and RandomForestRegressor models proved to be particularly efficient, with the former achieving an $R^2$ Score of 0.86 when predicting HTTP request rates and the latter reducing the Mean Squared Error (MSE) by 2.06% for memory usage predictions when compared to traditional monitoring methods.

These advances highlight the models' ability to identify crucial patterns and anticipate anomalies with considerable accuracy, enabling more agile and informed interventions. However, challenges such as the need for fine-tuning models and improving training performance still persist. The complexity and computational cost of machine learning models demand special attention, indicating the need for ongoing research into optimization and efficiency.

Future work will explore strategies that can speed up the training process without compromising the accuracy of the models. This could include the application of more efficient algorithms, the use of specialized hardware, and data dimensionality reduction techniques. In addition, emphasis will be placed on implementing auto-tuning mechanisms that can simplify the selection of hyperparameters, making predictive models not only more agile but also accessible for wider adoption in IT production environments. Furthermore, modifying the application to be able to run more than one application on different servers is also mapped out future work.

These future guidelines aim to strengthen the proposition that integrating machine learning into observability is a technical enhancement that can take IT systems management to a new level of proactivity and resilience.

# Declarations

## Acknowledgements

A special thanks to Chris Richardson for his effort in sharing the survey within the microservices community, helping to ensure a diverse and meaningful dataset.

## Funding

This study was financed, in part, by the following agencies: São Paulo Research Foundation (FAPESP), Process Numbers #2015/18808-0, #2022/14503-3; and National Council of Scientific and Technological Development (CNPq), Grant 420025/2023-5.

## Authors' Contributions

All authors equally contributed to the survey design and execution, literature review, data analysis, result discussion, and manuscript writing and revision.

## Competing interests

The authors declare that they have no competing interests.

## Availability of data and materials

`https://github.com/viniciusjssouza/dependable-microservices-paper`.

# References


Al-Qudah, Z., Rabinovich, M., and Allman, M. (2010). Web timeouts and their implications. In *International Conference on Passive and Active Network Measurement*, pages 211–221. Springer. DOI: 10.1007/978-3-642-12334-4$_2$2.

Amazon (2020). Amazon kinesis - easily collect, process, and analyze video and data streams in real time. Available at:`https://aws.amazon.com/kinesis/`.

Amiri, Z., Heidari, A., Navimipour, N. J., and Unal, M. (2023). Resilient and dependability management in distributed environments: A systematic and comprehensive literature review. *Cluster Computing*, 26(2):1565–1600. DOI: 10.1007/s10586-022-03738-5.

Apache (2022). Welcome to apache avro. Available at:`https://avro.apache.org/`.

Apache (2023). Apache kafka - a distributed streaming platform. Available at:`https://kafka.apache.org/`.

Assad, M., Meiklejohn, C. S., Miller, H., and Krusche, S. (2024). Can my microservice tolerate an unreliable database? resilience testing with fault injection and visualization. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 54–58. DOI: 10.1145/3639478.3640021.

Avizienis, A., Laprie, J. ., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33. DOI: 10.1109/TDSC.2004.2.

Bangui, H., Rossi, B., and Buhnova, B. (2022). A conceptual antifragile microservice framework for reshaping critical infrastructures. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 364–368. IEEE. DOI: 10.1109/ICSME55016.2022.00040.

Barr, J., Narin, A., and Varia, J. (2011). Building fault-tolerant applications on AWS. Available at:`https://media.amazonwebservices.com/AWS_Building_Fault_Tolerant_Applications.pdf`.

Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., and Rosenthal, C. (2016). Chaos engineering. *IEEE Software*, 33(3):35–41. DOI: 10.1109/MS.2016.60.

Belkhiri, A., Shahnejat Bushehri, A., Gohring de Magalhaes, F., and Nicolescu, G. (2023). Transparent trace annotation for performance debugging in microservice-oriented systems (work in progress paper). In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, pages 25–32. DOI: 10.1145/3578245.3585030.

Belshe, M., Peon, R., and Thomson, M. (2015). Hypertext transfer protocol version 2 (http/2). Available at:`https://www.rfc-editor.org/rfc/rfc7540?utm_source=localhost%3A8080`.

Bento, A., Correia, J., Filipe, R., Araujo, F., and Cardoso, J. (2021). Automated analysis of distributed tracing: Challenges and research directions. *Journal of Grid Computing*, 19(1):9. DOI: 10.1007/s10723-021-09551-5.

Beyer, B., Jones, C., Petoff, J., and Murphy, N. R. (2016). *Site Reliability Engineering: How Google Runs Production Systems*. " O'Reilly Media, Inc.". Book.

Bi, T., Pan, Y., Jiang, X., Ma, M., and Wang, P. (2022). Vecrosim: A versatile metric-oriented microservice fault simulation system (tools and artifact track). In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 297–308. IEEE. DOI: 10.1109/ISSRE55969.2022.00037.

Blohowiak, A., Basiri, A., Hochstein, L., and Rosenthal, C. (2016). A platform for automating chaos experiments. In *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 5–8. DOI: 10.1109/ISSREW.2016.52.

Bogner, J., Fritzsch, J., Wagner, S., and Zimmermann, A. (2019). Microservices in industry: insights into technologies, characteristics, and software quality. In *2019 IEEE international conference on software architecture companion (ICSA-C)*, pages 187–195. IEEE. DOI: 10.1109/ICSA-C.2019.00041.

Brooker, M. (2019). Timeouts, retries, and backoff with jitter. Available at:`https://d1.awsstatic.com/builderslibrary/pdfs/timeouts-retries-and-backoff-with-jitter.pdf`.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57. DOI: 10.1145/2890784.

Camilli, M., Guerriero, A., Janes, A., Russo, B., and Russo, S. (2022). Microservices integrated performance and reliability testing. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, pages 29–39. DOI: 10.1145/3524481.3527233.

Chen, Y., Yan, M., Yang, D., Zhang, X., and Wang, Z. (2022). Deep attentive anomaly detection for microservice systems with multimodal time-series data. In *2022 IEEE international conference on web services (ICWS)*, pages 373–378. IEEE. DOI: 10.1109/ICWS55610.2022.00062.

Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78. DOI: 10.1145/102792.102801.

Di Francesco, P., Lago, P., and Malavolta, I. (2018). Migrating towards microservice architectures: an industrial survey. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 29–2909. IEEE. DOI: 10.1109/ICSA.2018.00012.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham. DOI: 10.1007/978-3-319-67425-4$_1$2.

Fielding, R. T. and Taylor, R. N. (2000). *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine. Available at:`https://ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

Fowler, M. (2014). Microservices: a definition of this new architectural term. Available at:`https://martinfowler.com/articles/microservices.html`.

Fowler, S. J. (2016). *Production-Ready Microservices:*

*Building Standardized Systems Across an Engineering Organization*. O'Reilly Media, Inc, Sebastopol, CA, USA. Book.

Frank, S., Wagner, L., Hakamian, A., Straesser, M., and van Hoorn, A. (2022). Misim: A simulator for resilience assessment of microservice-based architectures. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 1014–1025. IEEE. DOI: 10.1109/QRS57517.2022.00105.

Gan, Y., Liang, M., Dev, S., Lo, D., and Delimitrou, C. (2021). Sage: practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151. DOI: 10.1145/3445814.3446700.

Gazzola, L., Goldstein, M., Mariani, L., Segall, I., and Ussi, L. (2020). Automatic ex-vivo regression testing of microservices. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, pages 11–20. DOI: 10.1145/3387903.3389309.

Ghofrani, J. and Bozorgmehr, A. (2019). Migration to microservices: Barriers and solutions. In *International Conference on Applied Informatics*, pages 269–281. Springer. DOI: 10.1007/978-3-030-32475-9$_2$0.

Ghofrani, J. and Lübke, D. (2018). Challenges of microservices architecture: A survey on the state of the practice. *ZEUS*, 2018:1–8. Available at:`https://www.researchgate.net/publication/328216639_Challenges_of_Microservices_Architecture_A_Survey_on_the_State_of_the_Practice`.

Gill, P., Jain, N., and Nagappan, N. (2011). Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 350–361. DOI: 10.1145/2018436.2018477.

Grohmann, J., Straesser, M., Chalbani, A., Eismann, S., Arian, Y., Herbst, N., Peretz, N., and Kounev, S. (2021). Suanming: Explainable prediction of performance degradations in microservice applications. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 165–176. DOI: 10.1145/3427921.3450248.

Gu, S., Rong, G., Ren, T., Zhang, H., Shen, H., Yu, Y., Li, X., Ouyang, J., and Chen, C. (2023). Trinityrcl: Multigranular and code-level root cause localization using multiple types of telemetry data in microservice systems. *IEEE Transactions on Software Engineering*, 49(5):3071–3088. DOI: 10.1109/TSE.2023.3241299.

Guo, X., Peng, X., Wang, H., Li, W., Jiang, H., Ding, D., Xie, T., and Su, L. (2020). Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1387–1397. DOI: 10.1145/3368089.3417066.

Hammer, R. (2007). *Patterns for Fault Tolerant Software*. O'Reilly Media, Inc, West-sussex, Inglaterra. Book.

Harsh, V., Zhou, W., Ashok, S., Mysore, R. N., Godfrey, B., and Banerjee, S. (2023). Murphy: Performance diagnosis of distributed cloud applications. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 438–451. DOI: 10.1145/3603269.3604877.

Haselböck, S. and Weinreich, R. (2017). Decision guidance models for microservice monitoring. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 54–61. IEEE. DOI: 10.1109/ICSAW.2017.31.

Heger, C., van Hoorn, A., Mann, M., and Okanović, D. (2017). Application performance management: State of the art and challenges for the future. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 429–432. DOI: 10.1145/3030207.3053674.

Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M. K., and Sekar, V. (2016). Gremlin: Systematic resilience testing of microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 57–66. DOI: 10.1109/ICDCS.2016.11.

Homer, A., Sharp, J., Brader, L., Narumoto, M., and Swanson, T. (2014). *Cloud design patterns: Prescriptive architecture guidance for cloud applications*. Microsoft patterns & practices. Book.

Hrusto, A., Engström, E., and Runeson, P. (2022). Optimization of anomaly detection in a microservice system through continuous feedback from development. In *Proceedings of the 10th IEEE/ACM International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems*, pages 13–20. DOI: 10.1145/3528229.3529382.

Huang, T., Chen, P., and Li, R. (2022). A semi-supervised vae based active anomaly detection framework in multivariate time series for online systems. In *Proceedings of the ACM Web Conference 2022*, pages 1797–1806. DOI: 10.1145/3485447.3511984.

Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., and Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35. DOI: 10.1109/MS.2018.2141039.

Jhawar, R. and Piuri, V. (2017). Fault tolerance and resilience in cloud computing environments. In *Computer and information security handbook*, pages 165–181. Elsevier. DOI: 10.1016/B978-0-12-803843-7.00009-0.

Kim, M., Sumbaly, R., and Shah, S. (2013). Root cause detection in a service-oriented architecture. *SIGMETRICS Perform. Eval. Rev.*, 41(1):93–104. DOI: 10.1145/2494232.2465753.

Kitchenham, B. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Technical report, Department of Computer Science, University of Durham. Available at:`https://legacyfileshare.elsevier.com/promis_misc/525444systematicreviewsguide.pdf`.

Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.". Book.

Knoche, H. and Hasselbring, W. (2019). Drivers and barriers for microservice adoption–a survey among professionals in germany. *Enterprise Modelling and In-*

*formation Systems Architectures (EMISAJ)–International Journal of Conceptual Modeling: Vol. 14, Nr. 1*. DOI: 10.18417/emisa.14.1.

Kumar, A. (2014). Software architecture styles: A survey. *International Journal of Computer Applications*, 87(9). Available at:`https://www.researchgate.net/publication/263026263_Software_Architecture_Styles_A_Survey`.

Lamport, L. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. Book.

Lee, C., Yang, T., Chen, Z., Su, Y., and Lyu, M. R. (2023). Eadro: An end-to-end troubleshooting framework for microservices on multi-source data. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1750–1762. IEEE. DOI: 10.1109/ICSE48619.2023.00150.

Lee, W. S., Grosh, D. L., Tillman, F. A., and Lie, C. H. (1985). Fault tree analysis, methods, and applications - 2013; a review. *IEEE Transactions on Reliability*, R-34(3):194–203. DOI: 10.1109/TR.1985.522211.

łgorzata Steinder, M. and Sethi, A. S. (2004). A survey of fault localization techniques in computer networks. *Science of computer programming*, 53(2):165–194. DOI: 10.1016/j.scico.2004.01.010.

López, M. R. and Spillner, J. (2017). Towards quantifiable boundaries for elastic horizontal scaling of microservices. In *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*, pages 35–40. DOI: 10.1145/3147234.3148111.

Ma, M., Xu, J., Wang, Y., Chen, P., Zhang, Z., and Wang, P. (2020). Automap: Diagnose your microservice-based web applications automatically. In *Proceedings of The Web Conference 2020*, pages 246–258. DOI: 10.1145/3366423.3380111.

Maeda, K. (2011). Comparative survey of object serialization techniques and the programming supports. *International Journal of Computer and Information Engineering*, 5(12):1488–1493. Available at:`https://publications.waset.org/15057/comparative-survey-of-object-serialization-techniques-and-the-programming-supports`.

McCaffrey, C. (2015). The verification of a distributed system. *Queue*, 13(9):60:150–60:160. DOI: 10.1145/2844108.

Meiklejohn, C., Stark, L., Celozzi, C., Ranney, M., and Miller, H. (2022). Method overloading the circuit. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 273–288. DOI: 10.1145/3542929.3563466.

Meiklejohn, C. S., Estrada, A., Song, Y., Miller, H., and Padhye, R. (2021). Service-level fault injection testing. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 388–402. DOI: 10.1145/3472883.3487005.

Mogul, J. C. (1995). The case for persistent-connection http. *ACM SIGCOMM Computer Communication Review*, 25(4):299–313. DOI: 10.1145/217391.217465.

Molléri, J. S., Petersen, K., and Mendes, E. (2016). Survey guidelines in software engineering: An annotated review. In *Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–6. DOI: 10.1145/2961111.2962619.

Nasab, A. R., Shahin, M., Raviz, S. A. H., Liang, P., Mashmool, A., and Lenarduzzi, V. (2023). An empirical study of security practices for microservices systems. *Journal of Systems and Software*, 198:111563. DOI: 10.1016/j.jss.2022.111563.

Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. (2015). How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73. DOI: 10.1145/2699417.

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 1st edition. Book.

Newman, S. (2019). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Incorporated. Book.

Niedermaier, S., Koetter, F., Freymann, A., and Wagner, S. (2019). On observability and monitoring of distributed systems–an industry interview study. In *International Conference on Service-Oriented Computing*, pages 36–52. Springer. DOI: 10.1007/978-3-030-33702-$5_3$.

Nygard, M. T. (2018). *Release it!: Design and Deploy Production-Ready Software*. Pramatic Bookshelf, Raleigh, NC, 2 edition. Book.

O'Neill, V. and Soh, B. (2022). Orchestrating the resilience of cloud microservices using task-based reliability and dynamic costing. In *2022 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, pages 1–6. IEEE. DOI: 10.1109/CSDE56538.2022.10089320.

O'Neill, V. and Soh, B. (2023). Spot market cloud orchestration using task-based redundancy and dynamic costing. *Future Internet*, 15(9):288. DOI: 10.3390/fi15090288.

Padmanabhan, V. N., Ramabhadran, S., Agarwal, S., and Padhye, J. (2006). A study of end-to-end web access failures. In *Proceedings of the 2006 ACM CoNEXT conference*, pages 1–13. DOI: 10.1145/1368436.1368457.

Pai, G. J. and Dugan, J. B. (2002). Automatic synthesis of dynamic fault trees from uml system models. In *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.*, pages 243–254. DOI: 10.1109/ISSRE.2002.1173261.

Panda, A., Sagiv, M., and Shenker, S. (2017). Verification in the age of microservices. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 30–36. DOI: 10.1145/3102980.3102986.

Petersen, K., Vakkalanka, S., and Kuzniarz, L. (2015). Guidelines for conducting systematic mapping studies in software engineering: An update. *Information & Software Technology*, 64:1–18. DOI: 10.1016/j.infsof.2015.03.007.

Pitakrat, T., Okanović, D., van Hoorn, A., and Grunske, L. (2018). Hora: Architecture-aware online failure prediction. *Journal of Systems and Software*, 137:669–685. DOI: 10.1016/j.jss.2017.02.041.

Poltronieri, F., Tortonesi, M., and Stefanelli, C. (2021). Chaostwin: A chaos engineering and digital twin approach for the design of resilient it services. In *2021 17th International Conference on Network and Service Management (CNSM)*, pages 234–238. IEEE. DOI:

10.23919/CNSM52442.2021.9615519.

Poltronieri, F., Tortonesi, M., and Stefanelli, C. (2022). A chaos engineering approach for improving the resiliency of it services configurations. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–6. IEEE. DOI: 10.1109/NOMS54207.2022.9789887.

Potharaju, R. and Jain, N. (2013). When the network crumbles: An empirical study of cloud network failures and their impact on services. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–17. DOI: 10.1145/2523616.2523638.

Protocol-Buffers (2023). Protocol buffers. Available at:`https://developers.google.com/protocol-buffers/`.

Rajagopalan, S. and Jamjoom, H. (2015). App–bisect: Autonomous healing for microservice-based apps. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA. USENIX Association. Available at:`https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/rajagopalan`.

Richardson, C. (2019). *Microservices Patterns*. Manning, Shelter Island, NY, USA. Book.

Rostanski, M., Grochla, K., and Seman, A. (2014). Evaluation of highly available and fault-tolerant middleware clustered architectures using rabbitmq. In *2014 federated conference on computer science and information systems*, pages 879–884. IEEE. DOI: 10.15439/2014F48.

Rouf, R., Rasolroveicy, M., Litoiu, M., Nagar, S., Mohapatra, P., Gupta, P., and Watts, I. (2024). Instantops: A joint approach to system failure prediction and root cause identification in microserivces cloud-native applications. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, pages 119–129. DOI: 10.1145/3629526.3645047.

Salah, T., Jamal Zemerly, M., Yeun, C. Y., Al-Qutayri, M., and Al-Hammadi, Y. (2016). The evolution of distributed systems towards microservices architecture. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 318–325. DOI: 10.1109/ICITST.2016.7856721.

Salfner, F., Lenk, M., and Malek, M. (2010). A survey of online failure prediction methods. *ACM Computing Surveys (CSUR)*, 42(3):1–42. DOI: 10.1145/1670679.1670680.

Samir, A. and Pahl, C. (2019). Dla: Detecting and localizing anomalies in containerized microservice architectures using markov models. In *2019 7th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 205–213. IEEE. DOI: 10.1109/FiCloud.2019.00036.

Sarda, K., Namrud, Z., Rouf, R., Ahuja, H., Rasolroveicy, M., Litoiu, M., Shwartz, L., and Watts, I. (2023). Adarma auto-detection and auto-remediation of microservice anomalies by leveraging large language models. In *Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering*, pages 200–205. DOI: 10.5555/3615924.3615949.

Schermann, G., Schöni, D., Leitner, P., and Gall, H. C. (2016). Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies. In *Proceedings of the 17th International Middleware Conference*, pages 1–14. DOI: 10.1145/2988336.2988348.

Scrocca, M., Tommasini, R., Margara, A., Valle, E. D., and Sakr, S. (2020). The kaiju project: enabling event-driven observability. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, pages 85–96. DOI: 10.1145/3401025.3401740.

Sedghpour, M. R. S., Klein, C., and Tordsson, J. (2021). Service mesh circuit breaker: From panic button to performance management tool. In *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*, pages 4–10. DOI: 10.1145/3447851.3458740.

Sill, A. (2016). The design and architecture of microservices. *IEEE Cloud Computing*, 3(5):76–80. DOI: 10.1109/MCC.2016.111.

Simonsson, J., Zhang, L., Morin, B., Baudry, B., and Monperrus, M. (2021). Observability and chaos engineering on system calls for containerized applications in docker. *Future Generation Computer Systems*, 122:117–129. DOI: 10.1016/j.future.2021.04.001.

Soldani, J. and Brogi, A. (2022). Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Computing Surveys (CSUR)*, 55(3):1–39. DOI: 10.1145/3501297.

Stocker, M., Zimmermann, O., Zdun, U., Lübke, D., and Pautasso, C. (2018). Interface quality patterns: Communicating and improving the quality of microservices apis. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–16. DOI: 10.1145/3282308.3282319.

Taibi, D., Lenarduzzi, V., and Pahl, C. (2018). Architectural patterns for microservices: A systematic mapping study. Available at:`https://www.scitepress.org/PublishedPapers/2018/67983/67983.pdf`.

Tam, D. S. H., Liu, Y., Xu, H., Xie, S., and Lau, W. C. (2023). Pert-gnn: Latency prediction for microservice-based cloud-native applications via graph neural networks. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 2155–2165. DOI: 10.1145/3580305.3599465.

Tanenbaum, A. S. and Van Steen, M. (2007). *Distributed systems: principles and paradigms*. Prentice-Hall. Book.

Thönes, J. (2015). Microservices. *IEEE software*, 32(1):116–116. DOI: 10.1109/MS.2015.11.

Viggiato, M., Terra, R., Rocha, H., Valente, M. T., and Figueiredo, E. (2018). Microservices in practice: A survey study. *arXiv preprint arXiv:1808.04836*. DOI: 10.1109/MS.2015.11.

Vishwanath, K. V. and Nagappan, N. (2010). Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 193–204. DOI: 10.1145/1807128.1807161.

VMware (2023). *RabbitMQ - Messaging that just works*. Book.

Waseem, M., Liang, P., Shahin, M., Di Salle, A., and Márquez, G. (2021). Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, 182:111061. DOI:

10.1016/j.jss.2021.111061.

Wittig, M., Wittig, A., and Whaley, B. (2016). *Amazon Web Services in action*. Manning. Book.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media. Book.

Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740. DOI: 10.1109/TSE.2016.2521368.

Wu, H., Yu, S., Niu, X., Nie, C., Pei, Y., He, Q., and Yang, Y. (2023). Enhancing fault injection testing of service systems via fault-tolerance bottleneck. *IEEE Transactions on Software Engineering*, 49(8):4097–4114. DOI: 10.1109/TSE.2023.3285357.

Wu, L., Tordsson, J., Elmroth, E., and Kao, O. (2020). Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE. DOI: 10.1109/NOMS47738.2020.9110353.

Xie, R., Yang, J., Li, J., and Wang, L. (2023a). Impacttracer: root cause localization in microservices based on fault propagation modeling. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE. DOI: 10.23919/DATE56975.2023.10137078.

Xie, Z., Pei, C., Li, W., Jiang, H., Su, L., Li, J., Xie, G., and Pei, D. (2023b). From point-wise to group-wise: A fast and accurate microservice trace anomaly detection approach. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1739–1749. DOI: 10.1145/3611643.3613861.

Xie, Z., Xu, H., Chen, W., Li, W., Jiang, H., Su, L., Wang, H., and Pei, D. (2023c). Unsupervised anomaly detection on microservice traces through graph vae. In *Proceedings of the ACM Web Conference 2023*, pages 2874–2884. DOI: 10.1145/3543507.3583215.

Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., and Zhou, L. (2009). Modist: Transparent model checking of unmodified distributed systems. pages 213–228. Available at:`https://www.usenix.org/legacy/event/nsdi09/tech/full_papers/yang/yang.pdf`.

Yang, J., Guo, Y., Chen, Y., and Zhao, Y. (2023). Hirca: A hierarchy anomaly diagnosis framework based on causality and correlation analysis. *Applied Sciences*, 13(22):12126. DOI: 10.3390/app132212126.

Yang, T., Shen, J., Su, Y., Ling, X., Yang, Y., and Lyu, M. R. (2021). Aid: efficient prediction of aggregated intensity of dependency in large-scale cloud systems. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 653–665. IEEE. DOI: 10.1109/ASE51524.2021.9678534.

Yu, G., Chen, P., Li, Y., Chen, H., Li, X., and Zheng, Z. (2023). Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 553–565. DOI: 10.5281/zenodo.8276375.

Yu, Y., Manolios, P., and Lamport, L. (1999). Model checking tla+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer. DOI: 10.1007/3-540-48153-$2_6$.

Zang, Z., Wen, Q., and Xu, K. (2019). A fault tree based microservice reliability evaluation model. In *IOP Conference Series: Materials Science and Engineering*, volume 569, page 032069. IOP Publishing. DOI: 10.1088/1757-899X/569/3/032069.

Zhang, J., Ferydouni, R., Montana, A., Bittman, D., and Alvaro, P. (2021). 3milebeach: A tracer with teeth. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 458–472. DOI: 10.1145/3472883.3486986.

Zhao, C., Ma, M., Zhong, Z., Zhang, S., Tan, Z., Xiong, X., Yu, L., Feng, J., Sun, Y., Zhang, Y., *et al*. (2023). Robust multimodal failure detection for microservice systems. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5639–5649. DOI: 10.1145/3580305.3599902.

Zheng, L., Chen, Z., He, J., and Chen, H. (2024). Mulan: Multi-modal causal structure learning and root cause analysis for microservice systems. In *Proceedings of the ACM on Web Conference 2024*, pages 4107–4116. DOI: 10.1145/3589334.3645442.

Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., and Ding, D. (2018). Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, pages 1–1. DOI: 10.1109/TSE.2018.2887384.

Zhu, Y., Wang, J., Li, B., Zhao, Y., Zhang, Z., Xiong, Y., and Chen, S. (2024). Microirc: Instance-level root cause localization for microservice systems. *Journal of Systems and Software*, page 112145. DOI: 10.1016/j.jss.2024.112145.

Zo, H., Nazareth, D. L., and Jain, H. K. (2007). Measuring reliability of applications composed of web services. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, pages 278c–278c. IEEE. DOI: 10.1109/HICSS.2007.338.

# A Supporting technologies for dependable microservices

Table 9 presents the weblink references for each technology we mention in Section 3.4.4. Fig. 6, as in Jamshidi *et al*. [2018] we organize dependable microservices according to goals, countermeasure techniques, and technologies.

| | Technologies | Weblink |
|---|---|---|
| Library, Framework | ASP.NET Core | https://docs.microsoft.com/aspnet/core |
| | Eclipse MicroProfile | https://projects.eclipse.org/projects/technology.microprofile |
| | Hystrix | https://github.com/Netflix/Hystrix/wiki |
| | Micronaut | https://micronaut.io/ |
| | Resilience4j | https://resilience4j.readme.io/ |
| | Semian | https://github.com/Shopify/semian |
| | Spring | https://spring.io |
| | Spring Retry | https://docs.spring.io/spring-batch/docs/current/reference/html/retry.html |
| | Steeltoe | https://steeltoe.io/ |
| Platform | AWS ELB/ALB | https://aws.amazon.com/elasticloadbalancing/ |
| | Google Cloud LB | https://cloud.google.com/load-balancing |
| | Mesos | https://mesos.apache.org/ |
| | Kubernetes | https://kubernetes.io/ |
| | AWS S3 | https://aws.amazon.com/s3/ |
| | AWS RDS | https://aws.amazon.com/rds/ |
| | Google Cloud SQL | https://cloud.google.com/sql |
| | Google Storage | https://cloud.google.com/storage |
| Third-party Application | Apache Skywalking | https://skywalking.apache.org/ |
| | Chaos Monkey | https://netflix.github.io/chaosmonkey/ |
| | Chaos Platform | https://netflixtechblog.com/chap-chaos-automation-platform-53e6d528371f |
| | Coverit | https://scan.coverity.com/ |
| | Datadog | https://www.datadoghq.com/ |
| | Dynatrace | https://www.dynatrace.com/ |
| | Elastic Stack | https://www.elastic.co/elastic-stack |
| | FindBugs | https://findbugs.sourceforge.net |
| | Grafana | https://grafana.com/grafana/ |
| | Graylog | https://www.graylog.org/ |
| | Gremlin | https://www.gremlin.com |
| | Istio | https://istio.io/ |
| | Infer | https://fbinfer.com/ |
| | Jaeger | https://www.jaegertracing.io |
| | Linkerd | https://linkerd.io |
| | Litmus | https://litmuschaos.io |
| | Loki | https://grafana.com/oss/loki |
| | Mangle | https://vmware.github.io/mangle/ |
| | New Relic | https://newrelic.com |
| | PMD | https://pmd.github.io/ |
| | Prometheus | https://prometheus.io/ |
| | Rollbar | https://rollbar.com |
| | Sentry | https://sentry.io/ |
| | Simian Army | https://github.com/Netflix/SimianArmy |
| | Spinnaker | https://spinnaker.io/ |
| | Zipkin | https://zipkin.io/ |
| Tool | TLA+ | https://lamport.azurewebsites.net/tla/tla.html |
| | TLC | https://github.com/tlaplus/tlaplus |
| | MoDIST | https://www.microsoft.com/en-us/research/project/modist-transparent-model-checking-of-unmodified-cloud-systems/ |
| | Spin | https://spinroot.com |

**Table 9.** Example of dependability technologies for microservices.

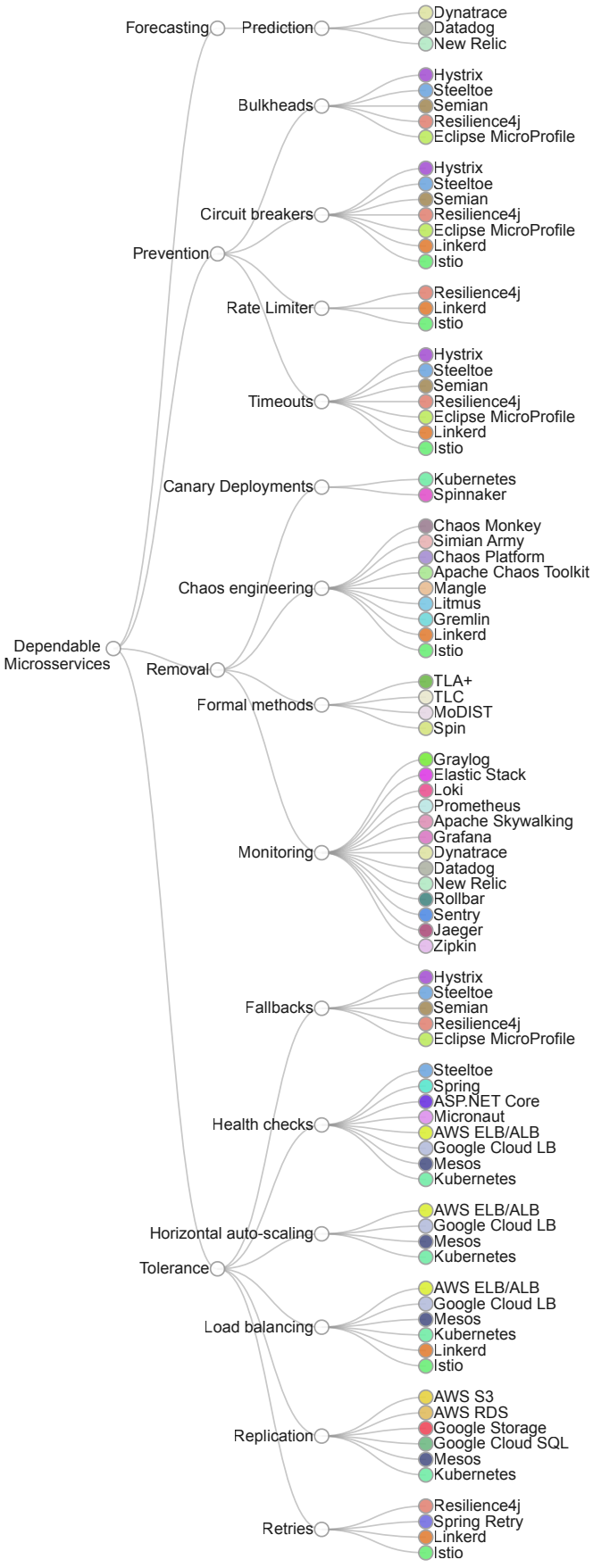


**Figure 6.** A big picture of dependable microservices, goals of dependability, countermeasure techniques, and technologies.