An Empirical Study on Challenges of Event Management in Microservice Architectures

RODRIGO LAIGNER, University of Copenhagen, Denmark

ANA CAROLINA ALMEIDA*, State University of Rio de Janeiro, Brazil

WESLEY K. G. ASSUNÇÃO, North Carolina State University, USA

YONGLUAN ZHOU, University of Copenhagen, Denmark

Microservices emerged as a popular architectural style over the last decade. Although microservices are designed to be self-contained, they must communicate to realize business capabilities, creating dependencies among their data and functionalities. Developers then resort to asynchronous, event-based communication to fulfill such dependencies while reducing coupling. However, developers are often oblivious to the inherent challenges of the asynchronous and event-based paradigm, leading to frustrations and ultimately making them reconsider the adoption of microservices. To make matters worse, there is a scarcity of literature on the practices and challenges of designing, implementing, testing, monitoring, and troubleshooting event-based microservices.

To fill this gap, this paper provides the first comprehensive characterization of event management practices and challenges in microservices based on a repository mining study of over 8000 Stack Overflow questions. Moreover, 628 relevant questions were randomly sampled for an in-depth manual investigation of challenges. We find that developers encounter many problems, including large event payloads, modeling event schemas, auditing event flows, and ordering constraints in processing events. This suggests that developers are not sufficiently served by state-of-the-practice technologies. We provide actionable implications to developers, technology providers, and researchers to advance event management in microservices.

CCS Concepts: • **Computer systems organization** → **Distributed architectures**; • **Software and its engineering**;

Additional Key Words and Phrases: microservice, asynchronous, event, streams, pubsub, decoupling, event-driven architecture, eda

1 INTRODUCTION

The emergence of cloud computing as a paradigm for large-scale deployment of services has prompted industry practitioners to rethink how applications are architected and deployed [5]. In particular, we witness a growing adoption of *microservice architectures* [130]. Microservice architectures promote designing components as independent building blocks that are deployed independently and interact with each other via network protocols [41]. This architectural style

---

brings more flexibility for the development and deployment of large systems, in contrast to conventional architectures, in which software components are strongly coupled with each other and deployed as a whole unit [73].

The main benefit of microservices is that development teams can develop and operate their services independently [120]. Microservices should be designed as self-contained software units that operate autonomously [73]. To realize business capabilities, microservices need to communicate, which ends up creating dependencies among their data and functionalities [59]. To manage such dependencies while maximizing decoupling, developers often rely on asynchronous event-based communication between microservices, instead of using traditional remote procedure calls [56, 58, 78]. The rationale is that, through events, producers and consumers are not directly coupled. For instance, producer microservices can trigger operations in other microservices or communicate their own state updates through events. Such an approach may positively influence the application's ability to evolve over time, limit the propagation of faults, and favor increased scalability of individual components [56, 83].

Event-driven architecture has been rapidly gaining industry popularity [15]. However, despite the benefits of an event-driven design, developers frequently report challenges in managing events in industrial settings [20, 35, 44, 58, 59, 75, 76, 78, 97, 98, 101, 102, 125]. Not surprisingly, it is common to encounter developers seeking support on how to implement event-driven microservices appropriately, such as this comment found on Stack Overflow (a popular Questions&Answers forum) [28]: *"Implementing an eventually consistent distributed architecture has turned out to be a pain. There are tons of blog posts telling stories about how to do it, but not showing (code) how to actually do it. One of the aspects I'm suffering is having to deal with manual retries of the messages when they haven't been acknowledged."* To make matters worse, in our study, we observed that there is a scarcity of literature on the practices and challenges that the adoption of an asynchronous paradigm brings to the designing, implementing, testing, monitoring, and troubleshooting of microservices. The challenges of managing communication in asynchronous and event-based systems have been explored in the fields of databases and distributed systems [17, 115], but this knowledge is not widely disseminated among developers [13, 59], making them sometimes try to "reinvent the wheel" when addressing common challenges. This limited understanding of existing practices and how to address challenges in architectures based on asynchronous communication leads to frustrations and ultimately makes companies reconsider the adoption of microservices.[1]

There has been extensive work investigating the adoption and implications of microservice architectures [9, 34, 59, 68, 71, 89, 114, 118, 119, 127, 129]. However, these pieces of work mostly focus on REST (representational state transfer) architectural style, describe the benefits of using microservices, or characterize data management and security practices without a holistic view of the practices and challenges. Furthermore, studies target specific companies or a limited number of practitioners, reducing the generalizability of their findings. To the best of our knowledge, no work has systematically investigated the practice and challenges brought about by an *event-driven* design in microservice architectures. As a result, although *event-driven* microservices form an important portion of microservice deployments today [59, 119], it is unclear what particular challenges developers face when developing and operating them. The need to address these issues is even more relevant when we consider studies suggesting there is a tension between the use of asynchronous microservice designs and application safety [59].

This work aims to characterize the practices and challenges faced by developers when adopting event-based microservice architectures. More specifically, we first investigate the practices adopted by developers to manage events in microservice architectures. We aim to comprehend current technological trends and common implementation patterns related to the adoption of event management. We also seek to identify the functional and non-functional

---

[1]Recent empirical studies have reported cases of companies moving from microservices back to monoliths [67, 106].

requirements that are met or supported by implementing event management in microservices. Secondly, we want to identify recurrent challenges faced by developers when managing events in microservice architectures. Focusing on these two perspectives, we can equip developers with knowledge of existing practices for event management in microservices and make them aware of the challenges they will face when developing asynchronous and event-based microservices.

To achieve our goal, we perform, to the best of our knowledge, the first empirical study to comprehend the practice and identify the challenges in managing events in microservice architectures from developers' perspective. To this end, our study is based on repository mining [45]. We mine and analyze relevant questions from Stack Overflow (SO),[2] one of the most popular Questions&Answers forums for developers who seek technical advice or assistance [19]. We collect more than 8,000 SO questions associated with managing events in microservice architectures, ensuring the diversity, representativeness, and quality of our dataset. Via a mix of keyword analysis and manual examination of tags, questions, and related responses (i.e., posts), we collect the most popular patterns and non-functional and functional requirements mentioned in the context of event management in microservices. Then, we randomly sample 624 relevant SO questions for manual analysis to identify the challenges. We carefully extract the challenges behind each question through a peer-reviewed process. We classify the uncovered challenges into different properties of distributed systems and functional and non-functional requirements. These ultimately represent key areas that microservice developers struggle with.

In the quantitative analysis, we observe that event management has been gaining increasing attention in practice, indicating the emergence of this architectural paradigm and the timeliness of this study. Results show the popularity of patterns and requirements related to achieving data consistency and loose coupling. This suggests microservice developers seek to strike a better balance between consistency and decoupling in their microservice architectures by employing asynchronous events, especially in computations that span microservices. Furthermore, we observe that developers report different categories of patterns in their implementations. For instance, observability (e.g., *distributed tracing*), performance (e.g., *circuit breaker*), and security (e.g., *access token*) patterns are reported being applied in questions involving event management.

In the qualitative analysis, we find that microservice practitioners face a myriad of challenges in managing events. While events are supposed to maximize the decoupling of microservices [38, 58], developers often surprisingly necessitate synchronizing events for correct event processing and coordinating microservices to allow for software evolution, contradicting the alleged benefits of an event-based architecture [39, 83]. Besides, although events can serve as natural progress markers of microservices [39, 56], we observe that developers find little benefit in using events to monitor and troubleshoot microservices. In particular, developers have a hard time tracking down the result of their computations, which often span a network of dependent microservices and exhibit additional difficulties in replaying past events for debugging purposes. Safeguarding security properties is also an emerging challenge in event-based microservice architectures. Developers struggle to synchronize event management and security technologies to ensure that only events generated by secured channels are processed and that only authenticated microservices can consume them.

This paper's contributions are manifold. We evidence that microservice developers are insufficiently served by state-of-the-practice technologies, including messaging systems, frameworks, databases, and cloud providers. They end up implementing their own ad hoc solutions to fulfill their requirements, which ultimately leads to errors and bugs, rendering adoption of a microservice architecture frustrating. Characterizing this emerging practice well informs

---

[2]https://stackoverflow.com/

general software developers, especially those unfamiliar with the challenges of designing distributed systems, about the hidden dangers of asynchronous, event-based microservice designs. We derive actionable implications for each class of event management challenges to researchers and message, database, cloud, and framework providers. These relate but are not limited to, devising novel microservice programming models, event management architectures, tools, and methods to advance the state-of-the-art practice in managing events in microservice architectures, making this a timely and valuable study. Furthermore, the results of our work can benefit different stakeholders involved in the development and management of event-driven microservice architectures:

- *For practitioners:* the challenges reinforce the importance of reliable sources of information and the need for careful analysis of the documentation of the technologies they adopt. Practitioners should question gray literature with general and high-level instructions that often do not embrace the complex nature of problems faced in production settings. Practitioners should not overlook problem-prevention measures, which include running, testing, and deploying their applications under the expected workload prior to production. Besides, developers must be cautious about "reinventing-the-wheel" approaches, since for every possible design attempt, there are reports of previous attempts.
- *For researchers:* software engineering, system engineering, and database communities must work closely to each other. Leveraging the knowledge from other fields can help developers address their challenges. Software engineering researchers may find no appropriate solutions for their problems in tools from system engineering, requiring only appropriate interfaces while safeguarding the application. Also, researchers must embrace/meet practitioners' expectations that were analyzed and discussed in the paper, solving real-world problems without focusing on "in vitro" experiments.
- *For tool builders:* documentation must pair up with/highlight the problems faced in practice. They should also provide actions that can be taken to mitigate the shortcomings of the existing tools. Furthermore, vendors must be clear about what requirements and guarantees cannot be properly achieved.
- *For educators:* we advise incorporating distributed systems concepts in software engineering courses, in particular algorithms and systems guarantees and their relationship with modern application development in the cloud. Yet, training students on how event management relates and makes its presence in modern application architectures, such as serverless, microservices, SaaS, to name some.

In addition, the dataset used in this study is made available [3] as an additional contribution to the research community, allowing other researchers to further investigate the theme.

The remainder of this work is organized as follows: Section 2 describes the role of events in modern microservice architectures and typical issues developers encounter. In Section 3, we present the methodology employed in this work. Section 4 presents the state of practice. Section 5 presents the specific challenges microservice developers encounter in managing events and their implications for the research community and industry providers. Section 6 discusses the threats to validity. Section 7 presents the related work. Lastly, Section 8 concludes this work.

2 BACKGROUND

To discuss the role events play in microservice architectures, we use Figure 1 to illustrate an ecommerce platform. We base on a popular microservice open-source project [4] that incorporates real-world event processing patterns.
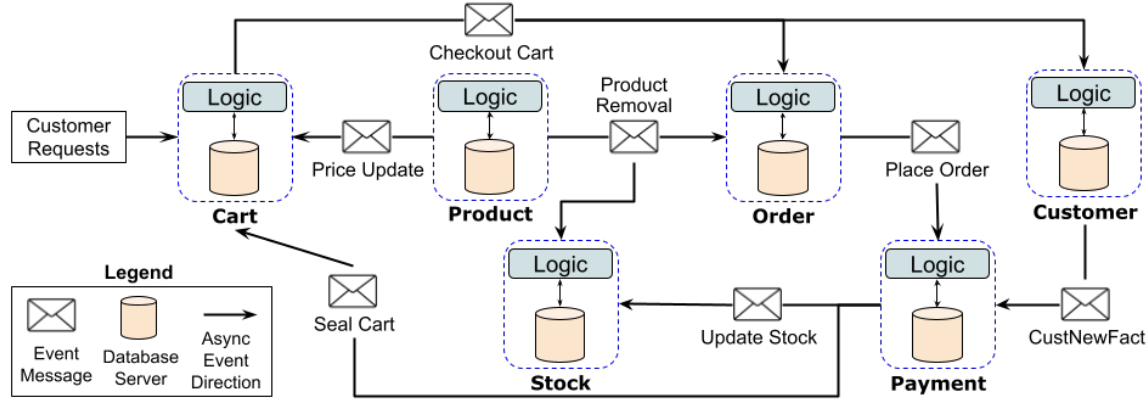
---

[3] https://zenodo.org/records/13149520

Fig. 1. Events in a microservice architecture

2.1 Context

Upon a customer's checkout request, the *Cart* microservice spawns the execution of a checkout through generating a `checkout_cart` event. The event is then consumed by the *Order* microservice, which processes the cart items and generates the `place_order` event. Next, the *Payment* microservice is responsible for processing the payment and generating the `update_stock` and `seal_cart` events for downstream consumption by the *Stock* and *Cart* microservices, respectively.

It is noteworthy that, in synchronous communication paradigms, such as through REST APIs and RPCs, the requester microservice gets blocked until the request terminates in the remote microservice and the response is received. It is natural to deduce that such blocking times can grow arbitrarily when business transactions traverse several microservices. In the example above, *Cart* would necessarily wait until *Payment* terminates the payment processing. Through the asynchronous paradigm, though, once the `checkout_cart` event is generated, *Cart* microservice is free to allocate computational resources (e.g., threads, CPUs, and memory) to serve other requests. This usually favors higher efficiency in event processing [72].

As exhibited in Listing 1, the communication through events is only possible because events carry semantic data that is used by microservices in distinct ways [39]. For instance, although the payment data is sent by the customer in a checkout request, such data is only used by the *Payment* microservice. That requires both *Cart* and *Order* microservices to pass along payment data in the `checkout_cart` and `place_order` events, respectively.

```
product_update:{ product_id, type, description, ... }
price_update:{ product_id, old_price, new_price, ... }
checkout_cart:{ customer_id, items:[{ product_id, qty, .. }],
    card_num, exp_date, address, discount }
update_stock:{ product_id, qty, timestamp, ... }
```
Listing 1. Example of application-generated event types

2.2 Problem Statement

Besides the apparent attractiveness of a decoupled design, managing events appropriately renders challenges. As events are generated asynchronously, that is, producers do not wait for consumers' reception, developers must account for the anomalies that possibly arise in event delivery, as described by the microservice adopter Nubank [75]:

> "you have late-arriving events, you can make a payment that we receive that should be credited on Friday, we need to time travel back, re-play the events and figuring out the balance that does not invalidate the invariants we have"

Besides, it is often reported that developers necessitate making sure anomalies in the event order does not lead to problems in their microservices, as explained by an Uber team [35]:

> "If it consumes an event that is not in sequence, our processing logic identifies the version mismatch and we retry the event a number of times."

Turning to our example scenario, if `price_update` events are processed in different orders, that can lead to incorrect product prices. Besides, supposing that the `product_removal` is not delivered to and processed by both *Order* and *Stock*, that leads the system to an incorrect state. In addition, if `place_order` is delivered more than once, that can lead to overcharging customers. An abnormal burst of events can also pose challenges to the system if consumers cannot cope with the arrival rate of events. In this work, we delve into these and several other types of challenges that affect event management in microservice architectures.

3 STUDY DESIGN

Although event-driven microservices are a popular architectural style in industry settings [56, 59, 83], existing studies mostly outlook the practical implications of adoption and the challenges brought about by an asynchronous and event-based design to microservices. Thus, the goal of this work is to *identify the practices adopted and challenges faced by practitioners developing software systems using event-driven microservice architectures.*

To achieve this goal, our study focus on answering two research question, as follows:

- **RQ1. What is the state of the practice on managing events in microservice architectures?** *Rationale*: Despite the popularity of event-driven microservice architectures, in the literature it is little known on why practitioners adopt this architectural style. Also, the literature is scarce on describing what are the practices adopted by them to manage events in microservice architectures, allowing practitioners to understand technological trends and popular implementation patterns associated with event management adoption. Thus, with this RQ, the insights we want to gain is threefold: (i) Understand the popularity trend of event management in microservice architectures; (ii) Collect the most popular patterns associated with event management adoption; (iii) Identify what functional and non-functional requirements are either enabled or facilitated by managing events in microservices.
- **RQ2. What challenges do practitioners encounter when managing events in microservice architectures?** *Rationale*: Once we understand the current practices on developing software systems using event-driven microservice architectures, we focus on identifying what particular challenges developers face when developing and operating them. In this question, we delve into the specific issues and impediments microservice developers face in practice while managing events.

To answer these two research questions, we use Stack Overflow (SO) [104], a popular question-and-answer online platform where practitioners seek technical assistance on issues they face in their day-to-day activities to understand the challenges of managing events in microservice architectures. Our methodology is similar to the ones adopted in previous studies [8, 121, 124] to collect, filter, and analyze questions. However, our focus is on event management in microservice architectures. We detail the procedure of our study below.

3.1 Data Collection

3.1.1 Downloading Stack Overflow dataset. We started by downloading the entire Stack Overflow dataset S_{all} from the official Stack Exchange Data Dump [103] available when we started this study (September 15, 2023). The dataset includes 23.199.461 questions dated from July 31, 2008 to December 31, 2022.[4] Every question in the dataset has title, body and a "tag" metadata, which denotes the topics on which the question lies.

3.1.2 Exploring initial tag set. We start with a general tag set to include as many relevant questions as possible. Thus, we use T_{ini} = {"microservice"}, resulting in 8369 questions, from which we performed an exploratory search. We observed that many questions were unrelated to event management, requiring significant effort to filter out. Also, we observed that relevant questions would mention the keywords "microservice" and "event" in either the title or body.

3.1.3 Filtering by relevant keywords. With the above insight in mind, we filtered the questions in T_{ini} that contain the keywords "microservice" and "event" in their title or body, leading to a total of 1407 questions (denoted as S_{rel}). We extracted 566 tags from S_{rel}, which we denote as T_{key}. Table 1 shows the distribution of questions per tag in S_{rel} (only the tags with five questions or more).

The first two authors jointly examined the candidate tags and observed that many of them relates to techniques, patterns, and technologies applied to managing events in microservice architectures, such as *event-driven architecture*, *pubsub*, *message queue*, and *event sourcing*. Even tags unrelated to event management (e.g., *authentication*, *logging*, and *database*) were often accompanied by event management-related tags, indicating possible correlated issues. Both observations gave us the confidence to proceed with further analysis.

3.2 Analyzing the State of the Practice

3.2.1 Patterns Trend. Based on a popular collection of patterns for microservice architectures [92], we extracted from the questions in S_{rel} the patterns for microservices mentioned by developers. It noteworthy we also included in the analysis the different users' posts (i.e., responses) for each question, totaling 3142 entries. We took extra care analyzing the dataset to embrace all possible forms of writing the same pattern (including typos), synonyms, and acronyms. For instance, we also identified the pattern "Command Query Responsibility Segregation" (CQRS) via "cqrs" or "CQRS." Section 4.1 answers RQ1(b).

3.2.2 Non-functional Requirements Trend. We also analyzed the most common non-functional requirements (NFRs) mentioned by practitioners. We use as reference the NFRs listed in Chung et al. [24]. We started searching keywords related to NFRs terms in S_{rel} (including posts) and incrementally introduced acronyms and synonyms. After introducing each keyword, we evaluated randomly selected related questions to validate whether the classification of the NFR fit the questions' context. We continued this process until exhausting the set of possible NFRs.

We understand that an NFR keyword search does not necessarily mean that such a requirement is part of the practitioners' problem. However, our analysis observed that such cases are often exceptions rather than rules, thus not undermining the analysis of the most pursued NFRs regarding event management in microservices. We respond to RQ1(c) in Section 4.2.

3.2.3 Functional Requirements Trend. We were also interested in understanding the specific functional requirements pursued by practitioners that are either enabled by or facilitated by managing events. Unlike NFRs, though, we noticed

---

[4]We excluded the year 2023 from our analysis as it was still a year in progress when this work was carried out

Table 1. Relevant tags extracted from Stack Overflow

Tag	#Questions	Tag	#Questions
microservice	895	messaging, saga	22
architecture	124	database	20
event-sourcing	90	masstransit, publish-subscribe	18
apache-kafka	78	event-handling, asynchronous	17
domain-driven-design	76	eventual-consistency, soa, distributed-transactions	15
cqrs	71	vert.x, distributed-computing	11
rabbitmq	54	distributed-system, authentication	10
event-driven	43	redis, google-cloud-pubsub	9
events	38	azure-service-fabric, event-bus, mes-sagebroker, authorization	8
event-driven-design	36	spring-cloud-stream, system-design, websocket, azureservicebus, socket.io, web-services	7
design-patterns	31	spring-kafka, bounded-contexts, database-design, event-based-programming, aws-lambda, google-cloud-platform, transactions, logging, amqp	6
message-queue	23	apache-kafka-streams, software-design, security, identityserver4, integration, <service>	5

that uncovering functional requirements often necessitated a thorough, detailed analysis to uncover the application scenario and, consequently, the functional requirement related to event management.

Relevance Filter. Given the high number of questions, manually examining all of them would demand a substantial amount of time, which could make reporting this study's evidence in a timely fashion impracticable. Therefore, at this point, we decided to apply a relevance filter. First, we use a relevance heuristic v to filter out questions with tags that are employed to a much lesser extent compared to others. As previous studies [8, 121], we considered only the tags whose v is higher than 0.005, leading to a set of 1298 questions and 93 associated tags.

$$v = \frac{\text{\# of questions with tag t in } S_{rel}}{\text{\# of questions in } S_{rel}}$$

Next, we focused on prioritizing the analysis of questions tagged with event management technologies (e.g., rabbitmq, kafka, masstransit) and techniques (e.g., cqrs, event-sourcing, event-handling) rather than questions tagged with technologies not directly related to event management (e.g., python, java, php, elasticsearch). Although the former also presents problems in the event management domain, thus part of our problem scope, we observe filtering by related event management tags led more often to questions revealing challenges within our study scope.

These two filters combined led to 53 tags (shown in Table 1) with a total of 925 questions. To proceed with the manual examination, we randomly sample 628 questions, which represents more than 2/3 of the relevant questions filtered.

Procedure. We adopt an open coding procedure [99] to analyze the sampled questions to pinpoint the specific functional requirements sought by practitioners inductively.

The first two authors, both of whom have years of software and data engineering experience, jointly participate in the manual examination of the questions. They analyze the sampled questions multiple times to familiarize themselves with them. In this process, many elements of the question were taken into consideration for inspection, including the title, body, code snippets, URLs, and the author's responses to other users' inquiries, which can ultimately clarify unclear points of the original problem statement (i.e., content in the body).

We observed that the user explicitly expressed the functional requirements being sought in most of the questions. For instance, in data replication cases, users would often mention that an event is generated based on some update (e.g., a user credit card score), so other microservices can also apply this change to their databases (see **S3** in Section 4.3).

However, in other cases, the specific functional requirement is not made explicit. For instance, different terms, like "consistency," are used to denote replicating data as a functional requirement, like the example as follows [122]:

> *Customer* and *Order* microservice both have customer details, though in *Order* microservice customer infos are striped to only required fields. I understand there is a way to maintain **consistency of data across microservices using events**.

Therefore, upon clarifying the functional requirement, the authors give terms to represent the requirements of the questions, using the terms used by developers whenever possible. We also found cases where a question contains more than a functional requirement. This way, we count the number of functional requirements independently of the number of questions. Section 4.3 answers RQ1(d).

3.3 Characterizing the Challenges

We use the same randomly sampled question set used above to analyze specific issues that developers report and characterize the challenges of managing events in microservice architectures.

Again, the first two authors adopt an open coding procedure and analyze all the elements of each question, including comments from users other than the author's question, to carefully extract insights about practitioners' challenges. The detailed procedure is as follows.

First, we ensured the practitioner's requirements were properly clarified. The reasoning is that we noticed the user often reports a problem faced subsequent to or in the context of expressing the non-functional and functional requirements sought. Since we already had the questions labeled with requirements through the previous methodology steps, our primary task remained to understand the challenges from the problem description and other question elements.

In many cases, the challenges could be identified by questions raised in the form of "how" or "what," as in other studies [121]. For example, one user asked [66]: "How can I process the event with the user credentials?" ; and another user inquired [116]: "What's maybe not a good idea in trying to recover the current state of your domain model by replaying an arbitrary set of your events ."

We also observed, though, in other cases, the users would not raise a question, but rather:
(a) express the willingness to achieve certain functionality [74]: "I want to create a third microservice that is responsible to join the data of *ProductService* and *StoreService*."
(b) describe the possible cause of a problem [51]: "In my system two different sources can cause creating specific type of event. [...] due to replication lag"
(c) inform a desired correctness criteria not being met [88]: "The end price should be 100 for that product, but sometimes these events are processed in random order."

In a few cases, the question's body lacks detailed information about the user's application scenario. In those cases, we searched for additional comments from the author's post made as a response to questions raised by other SO users. In the absence of further details about the user's problem scenario, the authors made the best effort to characterize the reported issue, jointly discussing whenever a disagreement was in place.

Second, once the problem statement was clarified, the authors started grouping similar problems into categories. The authors jointly iterated multiple times over the questions and categories. Whenever conflicts were observed in grouping the questions, a third arbitrator was introduced to discuss and reach a consensus. The third arbitrator has more than fifteen years of experience in cloud computing and data engineering. Lately, all questions come to an agreement and the final categories (i.e., the challenges) are confirmed by all the participants. Section 5 answers RQ2(a).

4 STATE OF THE PRACTICE (RQ1)

In this section, we describe the state of practice based on the discussions found in SO.

4.1 Patterns Trend

Table 2 exhibits the most recurrent patterns that appear in questions related to event management in microservices architectures [5]. In the top ten most mentioned patterns, four are directly related to event management (*messaging*, *event sourcing*, *domain event*, *cqrs*), one related to synchronous communication (RPC), two related to deployment patterns (*database per service* and *service-per-container*), and three related to typical microservice architectural patterns (*API gateway*, *service registry*, and *aggregate*).

We can observe that although event management patterns dominate the list, patterns unrelated to event management also appear substantially. For instance, remote procedure invocation, a synchronous communication technology, appears as the second most cited pattern. As events are processed asynchronously, the employment of synchronous communication mechanisms contrasts with the pursued benefits of events, which are often related to more efficient usage of computational resources and decoupling [39, 56, 58]. This trend is mainly due to the need for developers to reply to users the result of asynchronous computations, as exemplified in a developer's quote below:

> "How is it possible to send the results stored in the kafka topic back to the requesting WebClient_X?"

These challenges are further discussed in Section 5.1. Popular microservice patterns appearance in the top 10, namely, *API gateway* and *service registry*, can be justified by the need to allow web clients to initiate asynchronous computations and expose certain APIs to the external world, and to allow for service discovery , respectively. It is worth noting these are not concerns event management tackles in an event-driven architecture.

We also observed that the patterns *Database per Service* and *Service-per-Container* are often used synonymously by developers. The popularity of these two database deployment patterns suggests developers tend to avoid deployments that prescribe two or more microservices sharing the same database. For instance, the *messaging* pattern is only mentioned together with *Shared Database* is 12 occasions. We further discuss this trend in Section 4.2.

> **Highlight:** Adopting a shared database pattern would jeopardize the benefits of decoupling through events. The microservices would compete for computational resources of the machine (or container) hosting the database server, leading to decreased performance effects.

---

[5]For improved presentation, we show only those patterns appearing in 20 or more questions.

To aid our analysis, we grouped the patterns into pairs and count the number of times they are cited together in questions, as exhibited in Table 3. In general, the pairs' appearance is aligned with the individual numbers of Table 2. Event management patterns, such as *messaging*, *event sourcing*, *CQRS*, and *domain event*, appear substantially is pairs followed by *Remote Procedure Invocation* and *Database Per Service*. Other interesting insights are also confirmed, like the popularity of an authentication mechanism together with service discovery. Furthermore, the combination of pairs sheds light on the most common set of patterns considered in questions involving event management in microservices. *Messaging*, *event sourcing*, *Database per Service*, *CQRS*, *Service Registry*, and *Third Party Registration* are substantially mentioned in conjunction in questions, suggesting a pattern adoption trend.

To further aid our analysis, we grouped correlated patterns into categories shown in Figure 2. Patterns associated with data management form the most prominent category, followed by communication patterns, application architecture, support-service, deployment, security, and observability. We observe three subgroups within data management, including patterns associated with data consistency enforcement, database deployment, and data querying. Interestingly, in data consistency, we observe not only event-related patterns, such as *domain event* and *event sourcing*, but also *SAGA* and *eventual consistency*. That suggests developers also look for event management as an alternative to traditional, synchronous-based mechanisms for implementing consistency patterns, confirming the preliminary findings of [59].

> **Highlight:** Asynchronous events can enhance the performance of data consistency patterns by allowing for non-blocking interactions across microservices. That can lead to higher performance.

In the application architecture realm, event-driven architecture (EDA) is substantially mentioned, an expected trend given the scope of the study. However, Domain-Driven Design (DDD) also appears to a lesser extent. We found that only twelve questions mention EDA and DDD in conjunction, suggesting these two patterns are not popularly adopted together when it comes to event management in microservices.

> **Highlight:** As EDA often prescribes events trigger specific functions [39], our analysis suggests that this direct mapping between an event type and an application function (aka business logic) demotivates the modeling of extra modularity layers as prescribed by DDD.

We also observe the representativeness of patterns associated with deployment, such as *sidecar* and *service-per-container*, security with *access token*, and observability with *log aggregation*. The categories and related patterns exhibited in Figure 2 form the bulk of patterns cited by microservice developers. We explore further these phenomena in Section 5.

> **Finding 1:** There is a specific subset of patterns that appear frequently in event management questions. These mainly relate to database resource isolation, data consistency, and asynchronous messaging. Questions involving event management are also accompanied by mentions of patterns not directly related to event management, suggesting users often attempt to blend heterogeneous patterns in their deployments to solve cross-cutting concerns such as observability and security.

4.2 Non-Functional Requirements

Table 4 exhibits the extracted non-functional requirements from SO. We discuss the results along with key correlations found among NFRs as follows.

Consistency and Decoupling. Consistency appears to be the most cited NFR in event management questions in sync with the microservice patterns collected in the last section. A popular concern we noticed in this regard is the unmet

Table 2. Relevant patterns extracted from Stack Overflow

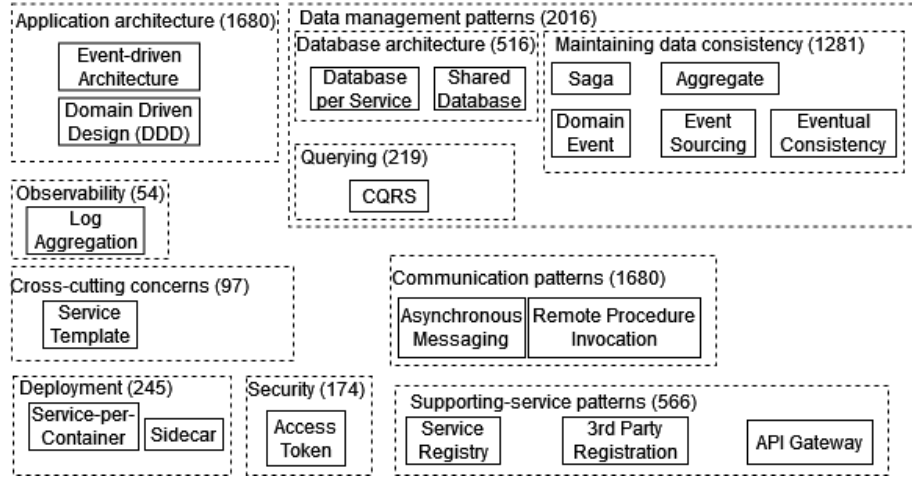| Pattern	# Posts	Pattern	# Posts
Messaging	1056	Shared Database	50
Remote Procedure Invocation	624	Circuit Breaker	48
Event Sourcing	537	Transactional Outbox	43
Database per Service	424	Domain-specific	42
API Gateway	249	Single Service per Host	39
Domain Event	225	Service-per-VM	38
Service Registry	221	CDC	38
Command Query Responsibility Segregation (CQRS)	219	Backend for frontend	32
Aggregate	207	Service per Team	32
Service-per-Container	192	Application Metrics	25
Access Token	174	Health Check API	25
Eventual Consistency	171	Externalized Configuration	24
Event-driven Architecture	151	Polling Publisher	23
Saga	141	Distributed Tracing	23
Domain Driven Design (DDD)	138	Self-contained Service	23
Service Template	97	Audit Logging	20
3rd Party Registration	96	Transaction Log Tailing	20
Log Aggregation	54	Multiple Service per Host	20
Sidecar	53		

Fig. 2. Most popular patterns in event management in microservices

expectation of certain events to arrive. For instance, one developer mentions a consistency problem observed whenever the delivery of events lags behind [16]:

> "When an **event isn't received** in the worker, the counters start to **"drift away" from the true** MySQL count.[...]"

Table 3. Relevant pairs of patterns mentioned in event management extracted from Stack Overflow

Pair of Patterns	# of Posts	Pair of Patterns	# of Posts
Remote Procedure Invocation & Messaging	278	Service Registry & Remote Procedure Invocation	69
Messaging & Event Sourcing	202	Messaging & CQRS	68
Remote Procedure Invocation & Event Sourcing	145	Service-per-Container & Remote Procedure Invocation	67
Messaging & Database per Service	132	Messaging & Aggregate	67
Remote Procedure Invocation & Database per Service	125	Remote Procedure Invocation & Domain Event	67
Event Sourcing & Database per Service	124	Messaging & Eventual Consistency	64
Event Sourcing & CQRS	122	SAGA & Messaging	60
Service Registry & Messaging	114	Domain Event & CQRS	58
Service Registry & 3rd Party Registration	90	Messaging & Access Token	57
Event Sourcing & Domain Event	85	Domain Event & DDD	56
Messaging & API Gateway	84	Remote Procedure Invocation & Event-driven Architecture	53
Remote Procedure Invocation & API Gateway	81	CQRS & Aggregate	51
Event Sourcing & Aggregate	81	Event Sourcing & API Gateway	51
Messaging & Domain Event	81	Eventual Consistency & Event Sourcing	50
Database per Service & CQRS	72	Service Template & Remote Procedure Invocation	50
Service-per-Container & Messaging	72	Remote Procedure Invocation & CQRS	50
Messaging & Event-driven Architecture	71		

Table 4. Relevant Non-Functional requirements extracted from Stack Overflow

Non-Functional Requirement	# of Posts
Consistency	312
Decoupling	303
Scalability	249
Performance	194
Modularity	170
Traceability	164
Security	154
Fault Tolerance	138
Load Balancing	109

Table 5. Relevant Non-Functional Requirement pairs extracted from Stack Overflow

Pair of NFR	# of Questions
Performance (24.23%) & Scalability (18.87%)	47
Decoupling (15.51%) & Scalability (18.87%)	47
Consistency (14.42%) & Decoupling (14.85%)	45
Consistency (12.5%) & Scalability (15.66%)	39

We observe that most developers express awareness that there is a natural delay to be expected. In a few cases, though, concerns over "how up to date" a given microservice is are implicitly raised, as exemplified by the same developer, as follows:

It is **expected** that there is a **consistency delay** for this type of data. How up-to-date the data is can even be figured out and included in the responses for stats data.

We further discuss the tension between the eventual delivery of events and application correctness in Section 5. Furthermore, we found that decoupling is the NFR most correlated with consistency. Analyzing the posts in which consistency and decoupling are mentioned together (45 posts), we observed that developers often identify trade-offs in pursuing the two NFRs in conjunction. The following extracted SO thread exemplifies this trend. A developer presents its application requirements starting with decoupling concerns [33]:

While each microservice generally will have its own data - certain **entities** are required to be **consistent across multiple services**.

I do not want shared database architecture, where a single DB manages the state across all the services. That **violates isolation and shared-nothing** principles.

And then highlights a possible mechanism to ensure cross-microservice consistency [33]:

"I do understand that, a microservice can publish an event when an entity is created, updated or deleted. All other microservices which are interested in this event can accordingly update the linked entities in their respective databases."

However, the developer realizes some inner drawbacks [33]:

"however it leads to a lot of careful and **coordinated programming effort across the services**. Can Akka or any other framework solve this use case? How?"

Another developer responds the thread acknowledging the tension between decoupling and consistency and the lack of principled solutions to the problem [12]:

I think there are 2 main forces at play here:

decoupling - that's why you have microservices in the first place and want a shared-nothing approach to data persistence

consistency - if I understood correctly you're already fine with eventual consistency

I don't know of any framework to do it out of the box, probably due to the many use-case specific trade-offs involved.

Performance and Scalability. Consistency and decoupling NFRs lead the number of post appearances in SO. However, taken together, they are outperformed by the pair performance and scalability. We observe developers express concerns on varied performance topics. For instance, both developers below face a problem when the event processing rate exacerbates the processing capacity of consumer microservices. We delve into performance issues on managing events in Section 5.3.

(DEV#1) Service A listens to a rabbit queue and sends http request to service B (which takes a couple of seconds). Both services scale based on the number of message in the queue. The problem is that the **requests from A to B are not balanced**. [...] That obviously causes **low performance and timeouts**. [61]

(DEV#2) My problem is the queue. **I can't get** an easily **scalable queue that guarantees ordering** of the messages. It actually guarantees "slightly out of order" with at-least once delivery [...] [21]

The same developer continues then:

> "But it turns out that using this solution, it will destroy **performance** when events are produced at high rates (I can use a visibility timeout or other stuff, the result should be the same)."

Fault tolerance. We also observe developers consider the adoption of event management technologies as a mechanism to increase fault tolerance in their microservice architectures. In the example below [100], a developer inquiries other users about whether Kafka can provide appropriate fault-tolerance support. We discuss about failures and their possible impact on microservices in Section 5.1.

> "I am working in a project that starts creating **independent deployable services**. The service we are creating should be **resilient** with an **24/7 uptime**."

> "In this case Kafka will be used as an **event system**. What do you think about the requirements and the usage of **Kafka to get a highly available and resilient** application?"

Finding 2: Consistency, decoupling, and performance appear as the most mentioned non-functional requirements. However, developers suggest there are trade-offs on properly meeting them in event-driven microservice architectures.

4.3 Functional Requirements

In this section, we discuss the most recurrent functional requirements sought by microservice developers on which event management is applied (denoted by **FR[0-N]**). Table 6 summarizes the number of functional requirements per question.

FR1. Propagation of state updates. We observe that the majority of questions contains a requirement related to propagating state updates in form of events. A developer summarizes this practice as follows [33]:

Table 6. Relevant Functional Requirements extracted from Stack Overflow

Functional Requirement	# of Questions
FR1. Propagation of state updates	78
FR2. Multi-microservice workflows	66
FR3. Data Integrity Maintenance	33
FR4. Replaying of events	22
FR5. Query Processing	20
FR6. Data replication	17
FR7. Cache management	12
FR8. Task Scheduling	5

> "A microservice can publish an event when an entity is **created, updated or deleted**. All other microservices which are interested in this event can accordingly **update the linked entities** in their respective databases."

Here we only take into account the questions where the developers' intention of propagating the events are either not expressed or not clear from the discussions. However, it is natural to deduce these state updates in form of events are used by other microservices to achieve other requirements, as we discuss next.

FR2. Multi-microservice workflows. We also observe a substantial number of developers reporting use cases where a business transaction require a composition of microservices [93] via events.

For instance, question 42140285 exhibits that an event generated by a microservice triggers an operation in another microservice [27]:

> "*Order* [**microservice**] receives an order request. It has to **store** the new Order ([record]) **in its database** and **publish a message** so that *Payment* **service** realizes it has to charge for the item"

Differently from the previous scenario though, we observe developers in this case have the expectation about the completion of the triggered computations.

In sum, in this functional requirement, the generation of an event act as a command, a call for action, for downstream microservices, manifesting the need to perform operations, often involving the operations in their private states.

FR3. Data Integrity Maintenance.

We also found questions that developers highlight how an event is used to maintain the integrity of a microservice's state. An example is provided as follows [112].

> "I have a web-api-endpoint that receives orders that an *OrderMS* is responsible to handle. When order is put[,] Inventory must be updated so **OrderMS** publish an event to subscribers [...] and **InventoryMS will update the inventory** due to it is subscribing to current event/message"

Another developer explains the intention to achieve functional dependency across microservices via events propagated [122]:

> "Customer and Order microservice both have customer details, though in Order microservice customer infos are striped to only required fields. I understand there is a way to maintain consistency of data across microservices using events."

FR4. Replaying of events. We observe that developers seek replaying (i.e., resending) past application-generated events in two cases:

(i) Synchronizing states. Microservice applications evolve over time. As new microservices are incorporated into the topology, it may be the case that these are required to synchronize with the current application state, as described by a developer [117]:

> "When I add **new service** to a set of already running services, I need upstream dependencies to send all the messages from the past to the new service so it could **align its state** with the one of the whole system."

(ii) Troubleshooting and Auditing. As the communication abstraction in event-driven microservices, it is natural that the re-execution of the event flow may assist in further understanding the application behavior, as explained by the following developers:

> (DEV #1) I want to **replay events** on an invoice where the I want **to see all actions done** by a specific employee on the balance. [82]

> (DEV #2) The intent for this pattern [(event sourcing)] is to provide an **audit trail of all events** that took place while the patient was in the hospital. [107]

FR5. Query processing. Developers tend to report the implementation of query processing operators at the application level. In this case, they implement queries that operate over the payload of subscribed events, as exemplified by the following SO question [74]:

> "I want to create a third microservice that is responsible to **join the data of ProductService and StoreService** in order to retrieve all the available products. Here, CQRS pattern seems like the best solution: I will create a materialized view and I will synchronize it using domains events published by the other 2 microservices."

FR6. Data replication. We observe that the propagation of state updates is also an enabler of data replication. By subscribing to state updates in the form of events, consumers can maintain their own view of the state managed by

other microservices without requiring to pull updates, thus avoiding the overhead entailed by synchronous calls (e.g., RPCs) [115].

For instance, in the following question [55], the developer explains that

> "Service A could raise an event whenever a new student is registering. Service B will consume the event and **stores student info in its [own] db**."

Based on our analysis, it is worth noting that not all cases of propagation of state updates will decidedly lead to data replication, but all cases of data replication via events necessarily require propagation of state updates.

FR7. Cache management. In a similar way to FR6, another reported scenario lies on maintaining caches based on subscribed events, as explained by a developer as follows [109].

> "When C starts up it needs to **load all of the current data from P into its cache**, and then subscribe to change notifications. (In other words, we want to synchronize data between the services.)"

FR8. Task Scheduling. We also found cases about events being used as abstractions to manage the life cycle of long-running jobs. As example scenarios, we highlight the following:

> "That job info is placed in a RabbitMQ message and sent off by the RabbitMQ Producer [..] A RabbitMQ Consumer receives message with the job info and calls the class that is responsible for Executing the **long running job**, the job status is updated to IN-PROGRESS" [79]

> "The idea is to have the REST API immediately post a message to a queue, with a background worker role picking up the message from the queue and **spinning up multiple backend tasks**, also using queues. REST API [..] generates a GUID and attach that as an attribute on the message being added to the queue" [111]

Finding 3: Event management is applied to fulfill varied functional requirements, such as communicating data updates, composing microservices functionalities, and processing queries, indicating the heterogeneous applicability of events.

---

5 EVENT MANAGEMENT CHALLENGES (RQ2)

Having discussed the state of practice of event management, we focus on the challenges developers face when managing events in their microservice deployments. Figure 3 summarizes the challenges we extracted from SO. The leaf nodes represent the specific challenges, whereas their parent nodes represent the categories to which a challenge belongs. For example, the *Performance* (5.3) category comprises three specific challenges: event processing overhead (CP1), large event payload (CP2), and fluctuating event rate (CP3). In total, our analysis led to five categories and sixteen specific challenges, indicating the heterogeneity of the problems microservice developers encounter when managing events. Next, we discuss and exemplify each specific challenge by their categories. Along the discussion, it is worth noting that we refer to messaging technology as any system that enables microservices to exchange events through publishing and subscription mechanisms.

5.1 Safety and Liveness

Safety and liveness are properties inherent to distributed systems [60]. While safety properties prescribe that nothing bad happens, liveness properties prescribe that something good eventually happens. For instance, a traditional example of safety property is found in the 2-Phase Commit (2PC) protocol [69]. In 2PC, the effects of the operations of a transaction
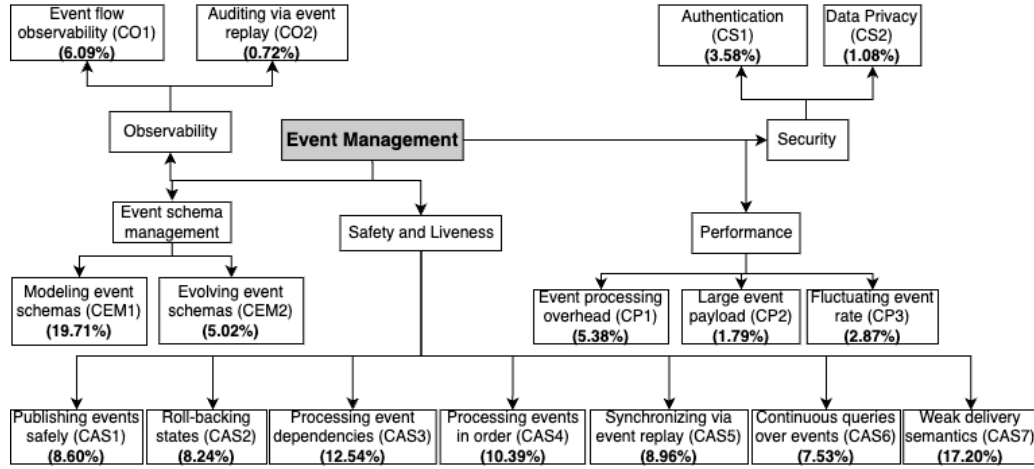
Fig. 3. Overview of challenges in managing events in microservice architectures

that cuts across nodes are only made available to subsequent transactions if all participating nodes agree. A common liveness guarantee example is eventual consistency [83]. In this consistency model, nodes eventually converge to the same outcomes (e.g., data item versions). When this is achieved, it is often unbounded, making it a weak consistency model.

In this section, we use these two properties to discuss related problems in the context of event management in microservice architectures. We reveal microservice practitioners' expectations when managing events, particularly when publishing, consuming, processing, reverting the effects of, and replaying events.

5.1.1 Publishing events safely (CAS1). Safety.

The functional requirements investigated in Section 4.2 highlight that generating events is the cornerstone of event-based microservice architectures. Events are often raised in response to local operations, typifying a causality relation. Therefore, failures in local operations must automatically withdraw the existence of a resulting event. However, we find that event's publishing semantics are not always crystal clear for microservice developers. We highlight as follows the most common types of issues using developers' own quotes:

(i) Developers seek to acknowledge whether the event published has reached out to the message broker, for instance [27]:

> (DEV#1) "This operation [(publishing the event)] is asynchronous and **ALWAYS returns true**, no matter if the broker is down. How can I know that the message has reached the broker?"

(ii) Developers wonder whether publishing an event as part of a transaction is possible, for instance [81]:

> (DEV#2) "[...] does actually spring-kafka support JTA transactions and would it be enough to wrap RDBMS and Kafka Producer into **@Transactional [(ORM annotation)] methods**?"

(iii) In other cases, developers wonder how to ensure atomicity semantics when publishing more than an event as part of an operation, like the user below [64]:

(DEV#3) "Let's say a command needs to append an event to both, the public and private user stream. **How can you make sure that both events have been appended?** Does the event store publish both, `SomeUserEventHapppendPrivate` and `SomeUsererEventHappendPublic`, to the event bus?"

(iv) Developers look for alternative ways to publish events out of the critical path of the application, such as asynchronously detecting database updates and publishing these as events, like the case below [94]:

(DEV#4) "Should I have some kind of **background timed process** which will scan events table and publish events to SQS? Can this be process within WebApi application (preferable), or **should this be a separate a process**?"

Challenges related to ensuring events are correctly published appear in 13.79% of `Safety&Liveness` challenges and reflect 8.60% of the total, indicating the representativeness of the problem.

Discussion: We observe practitioners seem unfamiliar with the guarantees provided by messaging technologies. As a result, they may fail to foresee the possible issues associated with ensuring consistency between two systems, in their case, the (producer) microservice and the message broker.

Recent Change-Data-Capture (CDC) tools like Debezium [6] soften this challenge by dedicating a system to capture updates in a microservice private state (e.g., relational tables), offloading developers from explicitly handling the publishing of events. Each update is then eventually delivered to consumer microservices. However, apart of the need to manage an additional system, when or whether this delivery is performed is unknown, which can lead to an additional challenge as we see in Section 5.1.7.

Furthermore, some database systems like Oracle [7] and PostgreSQL [8], offer mechanisms to publish events atomically as part of transactions. However, the benefits of state encapsulation would be jeopardized given all consumer microservices would depend on specific producer-managed databases, thus leaking the private microservice state abstraction.

5.1.2 Roll-backing states (CAS2). **Safety.**

Once an application-generated event is published (i.e., reach out to the message broker successfully), the event becomes available for consumption by other microservices, which in turn can consume and start processing the event. However, events may be consumed and processed by multiple microservices independently, which makes it challenging to track down the possible errors across multiple microservices and react accordingly. As a result, we observe that developers express uncertainties on how to appropriately deal with failures, such as the following cases:

(DEV #1) "[...] the state modification is actually a complex operation across multiple microservices using the saga pattern, which needs to be rolled back if something fails. [...] **How to cleanup the state if the service modified it, but failed in the end**, e.g. due to system shutdown?" [91]

(DEV #2) "When my Transaction is committed, normally I will dispatch `IntegrationEvent` (e.g. to the queue), but there is possibility that this **queue is down** as well, so previously just-committed **transaction has to be "reverted"**. How?" [85]

In this sense, many answers received for related questions suggest the use of compensating actions triggered via additional events; however, some developers acknowledge that these inherit the same properties of the originating problem, as summarized by the following quote:

---

[6]https://debezium.io
[7]https://www.oracle.com
[8]https://www.postgresql.org

(DEV#3) "If 3.a ([an operation triggered via event]) fails[,] a **compensation action** is performed; but **what about if it fails**?" [52]

We observe this challenge is significant in the context of `Safety&Liveness` challenges (13.22%), spanning across 8.24% of the total questions.

Discussion: When the event is published, the event producer loses control over downstream processing due to the independent failure of microservices. Similarly, consumers are unaware of which microservice produced the event stream to which they are subscribed. In this case, we found some developers design specific event streams to communicate failures in the workflow, as we discuss in Section 5.1.2.

However, publishing events that semantically represent the occurrence of a failure can also fail. That can lead "dirty" intermediate states (i.e., the effects of canceled operations) to remain exposed indefinitely. On the other hand, even though upstream microservices acknowledge a downstream failure, once the local operations are committed as part of a transaction in the database, it is often not possible to rollback the transaction. Thus, upon failure, it is unclear for developers how to properly revert those changes.

5.1.3 Processing event dependencies (CAS3). **Liveness.**

Events often carry a schema, or an event type, and are published in a stream (aka topic or queue) on which all events published must adhere to such schema. We observe an interesting trend in which practitioners attempt to match distinct but correlated events at event processing time. The trend is exemplified by a developer [43]:

(DEV#1) It is very straightforward to implement a [(event)] listener when **an action depends on one single event** ([stream]). [...] The problem arises when *OrderService* **has to wait for more than one event** [(from different streams)]

The developer contextualizes the case and expresses the problem then [43]:

(DEV#1) I am pooling both queues
`CREDIT_AVAILABLE_QUEUE` and
`INVENTORY_AVAILABLE_QUEUE`, and **both events has to be present** so I can finish an order. How can I coordinate so that *OrderService* sees both events as only one?

However, these dependencies across distinct events may require ordering guarantees from the messaging technology, which may not be present, as noted by another developer [21]:

(DEV#2) If `UserCreated` **comes after**
`ProductAddedToCart` the normal flow requires to **throw an exception** because the **user doesn't exist yet**. [...] So, the ordering problems are:
No order guaranteed across events from the same event stream.
No order guaranteed across events from the same ES [(event store)].
No order guaranteed across events from different ES (different services).

These ordering problem can ultimately lead to concurrency bugs in the application code, as noted by the former developer [43]:

(DEV#1) There is a minimal chance of receiving both events at the same time[,] generating **race conditions**.

We observe that this challenge appears substantially, spanning across 20.11& of `Safety&Liveness` challenges. Besides, it spans 12.54% of the total questions, highlighting the practical significance of this event processing pattern.

Discussion: Developers express some events are semantically related and thus cannot be processed independently. However, developers fall prey to weak ordering semantics across event streams. Another challenging factor is that the delivery of events can experience arbitrary delays due to network partitions, overload of computational resources, network jitter, and even the failure of producer microservices. These aspects together pose a challenge in enforcing event dependencies across distinct event streams.

This event processing characteristic may find resemblance with the so-called workflow patterns, posited for business processes and web service compositions [50] a few decades ago. Recent microservice frameworks, like Dapr [9], has included in their API support for some workflow patterns, such as fan-out/fan-in [30]. However, workflow patterns often require either a domain-specific language or a common framework where all microservices are implemented with, as well as the explicit specification of workflows, including event producers and consumers. These contrasts with the polyglot nature of microservices and the decoupling through events seek by microservice developers. Thus, it is an open question whether and how workflow patterns can mitigate the challenges of processing event dependencies.

5.1.4 Processing events in order (CAS4). **Safety.**

While analyzing the questions, we noticed that practitioners often expect events to be processed in order for every consumer. However, in many cases, this expectation is not met. Unlike the previous challenge, this challenge affects single event streams (i.e., all conforming to the same event type) rather than distinct events. A developer exemplifies the problem as follows.

> (DEV#1) Suppose a product is ordered and it is *id* 80, and a series of sequential update events fired from product service to order services for that particular product [...] **The end price should be 100 for that product, but sometimes these events are processed in random order** [88]

The developer expects that the product price observed by the *Order* microservice must eventually be in sync with the product's state in the producer (i.e., *Product* microservice). Therefore, any processing order not leading to a price of 100 as the final state is deemed incorrect. In some cases, practitioners acknowledge that an in-order processing guarantee cannot be enforced in consumers due to the uncertain semantics of producers:

> (DEV#2) Since each microservice has its own event table and asynchronous worker, **we cannot guarantee that events will be sent in the sequence** in which the corresponding state changes occurred in their respective microservices. [77]

This challenges appears in 16.67% of `Safety&Liveness` challenges and accounts for 10.39% of the total challenges observed, highlighting the difficulty of ensuring events are processed in the expected order.

Discussion: Although the majority of messaging systems guarantee the delivery of events from the same stream in order, such as Kafka through the concept of a topic partition [10], that does not exclude producers and consumers from still processing the events accounting for the expected processing order semantic. For example, for producers consisting of concurrent threads writing to the same partition, even though Kafka serializes the messages, the correct order that the consumer must process the messages is not well defined. Similarly, suppose multiple consumer threads pull messages concurrently from Kafka. In that case, if the order of messages across different pulls matters, it is complex and error-prone for general developers to reassemble the original stream order at the consumer side.

---

[9]https://docs.dapr.io

In most questions analyzed with this particular challenge, we could not identify whether the message broker delivers the messages out of order or the microservice processes the event arbitrarily (not accounting for the delivery order). As the quote below exemplifies [80], event processing semantics appear unclear to some microservice developers.

> (DEV#3) In [(omitted technology,)] **messages are best-effort ordering, still no idea of what they mean**. [...] Does it means that[,] giving n copies of a message[,] the first copy is delivered in order[,] while the others are delivered unordered compared to the other messages' copies? Or "more that one" could be "all"?

5.1.5 *Synchronizing states via event replay (CAS5).* **Safety.**

As described in Section 4.2, the need for replaying events often arises due to new microservices being introduced in the system. In this context, a developer explains an interesting challenge [84]:

> (DEV#1) Service "A" creates event message to inform other services about changes [...] **Newly introduced Service "D" also needs to replicate data** coming from service "A". Service "D" needs all historic data[.]

The developer then continues expressing the impedance found.

> ([...]) other services were already running for a while, and **service "A" only broadcasts new changes**. What would be the correct solution to populate newly added service with historic data?

Furthermore, as microservices supposedly fail independently, it may be necessary to replay events from the point in time the subscriber microservice(s) failed, as described by another developer [3]:

> (DEV#2) I'm wondering what should happen if one of these events can not be delivered due to an error [...] On republishing events, should all messages be republished to all topics or would **it be possible to only republish a subset**?

Challenges involving replaying events also appear significantly in `Safety&Liveness` challenges (14.37%), accounting for 8.96% of the total challenges, suggesting mechanisms for safely replaying events are strongly seek by developers but missing in practice.

Discussion: Along the analysis, we find that the need for replaying events arises not only from software evolution, including adding new microservices and migrating to a microservice architecture but also when recovering from failures and fixing the outcome of bugs. However, developers express uncertainties about how to proceed properly in the cases above and end up implementing several ad-hoc mechanisms at the application layer to fulfill these. As a result, these mechanisms often result in rework and create additional issues in the application.

5.1.6 *Continuous queries over events (CAS6).* **Liveness.**

We observe that practitioners commonly carry out continuous queries consisting of various query operators (e.g., filter, join, and aggregations) [126] over the payload of incoming events to extract valuable information. However, the nondeterminism of asynchronous events (i.e., the arrival of events may experience large delays or even never arrive) poses challenges in ensuring correct query processing results, as described by a developer [74]:

> (DEV#1) When I update a *Product* entity or *Store* entity there is no problem to sync the view because I already have data on it. But what about when I receive a `ProductCreated` event?

> This event has only information about product and nothing else, so `stock`, `store_name`, `store_address` **will be NULL**.

How can I save this event in my view? **Should I save incomplete data somewhere else** and update my view **when I will receive complete data**?

Furthermore, as software evolves over time, changing requirements can also undermine providing correct query results based on events, as exposed by another developer [49]:

(DEV#2) I have **new requirement** - calculating maximum all time temperature per sensor.

I have prepared **new microservice** that creates `KTable` ([Kafka table abstraction]) aggregating temperature (with max) grouped per sensor.

Simply deploying this microservice would be enough if input topic had infinite retention, **but now maximum would be not all-time**, as is our requirement. [...] How to design the solution? (67074772)

The impediment found above relates to the inner characteristic of stream processing engines, that necessarily rely on windows and notions of time to avoid querying all the historical data in order to provide fast responses [48]. Continuous queries over events appear frequently in Safety&Liveness challenges (12.07%) and corresponds to 7.53% of the total questions, highlighting the significance of this challenge.

Discussion: For over two decades, continuous queries and stream processing systems have been a mainstream research topic in the data management community [40], resulting in industry-strength solutions like Flink [10] and Kafka Streams [11]. These systems allow users to declare a query that is continuously updated based on incoming streams. However, we notice microservice developers often do not mention stream processing systems and prefer patterns like CQRS, not realizing that providing correct query results with this pattern incurs additional challenges. Besides implementing the query operators, developers must ensure that duplicate events (we further discuss in Section 5.1.7), crashes, and delays do not impact the query results.

Stream processing systems are designed to handle the gamut of problems that arise in maintaining continuous queries. We conjecture microservice developers are either unaware of or may find difficult to map their application logic to stream processing operators.

5.1.7 Weak delivery semantics (CAS7). **Liveness.**

Messaging technologies often guarantee at-least-once delivery by default, leaving the responsibility of achieving at-most-once or exactly-once delivery semantics to application developers [40]. However, by resorting to at-least-once delivery semantics, developers can face unpredictable challenges, as exemplified by the developer as follows [53]:

(DEV#1) I was able to collect the data [(published as events)] from another microservice, but I noticed that if I terminate the [(consumer)] process and run it again, I get the data back, PLUS another copy of the same data right under it. [...] **I got the payload sent to me twice, and I only want to see it once**.

As the developer's quote shows, a problematic aspect of at-least-once delivery is that events can be delivered more than once. That forces consumers to either make their microservices idempotent or fall prey to the undesired effects of duplicate event processing. The following microservice developer expresses another interesting case [29]:

(DEV#2) The only way I have **to know** that the **payment has been processed** by *Payment* service **is by expecting an answer event** (`Payment ok|failure`).

However, in the absence of an event that confirms that a payment has been processed, the developer comes up with its own customized solution:

---

[10]https://flink.apache.org
[11]https://kafka.apache.org/documentation/streams

> If it hasn't gotten an answer in some time, [that] **forces me to implement a retry mechanism** in the
> *Order* server, [that is,] retry with a new `Payment` event.

The developer ended up dealing with an unexpected outcome:

> [However,] this also **forces me to take care of duplicated messages** in *Payment* service in case they
> were actually processed but the answer didn't get to the *Order* service.

Another popular issue on resorting to events for asynchronous processing lies on the fact that computations are often initiated based on client requests. These online requests require timely response to users and challenges developers to match the eventual arrival of asynchronous events (i.e., the response of the client-triggered asynchronous computation) with the corresponding waiting client. A developer describes the problem below [6]:

> After publishing the event[,] my booking **service can't block the call and goes back to the client**
> (front end).

> How does my client app will have to check the status of transaction? Does it poll every couple of seconds?
> Since this is distributed transaction and **any service can go down and won't be able to acknowledge
> back**.

> In that case how do my client (front end) would know since **it will keep on waiting**.

Dealing with weak delivery semantics is the most popular challenge in `Safety&Liveness` category (27.59%). By appearing in 17.20% of total the questions, that evidences the difficulty of managing events under weak delivery semantics.

Discussion: The loss of events may be caused by many issues. Besides consumers crashing, during a rebalance in Kafka brokers, consumers can experience delays in event delivery [37]. Besides, the network can become unstable, leading to delays in network package deliveries. In an attempt to ensure the completeness of operations across microservices, developers often rely on custom-made, ad-hoc solutions, such as the retry mechanism shown above. However, retries can lead to the additional burden of dealing with duplicate events, which forces developers to implement additional mechanisms at the application layer to impede duplicate events from being processed, only exacerbating the problem.

> **Finding 4:** Developers encounter a myriad of challenges on ensuring events are processed correctly. These span
> the entire life-cycle of an event in a microservice architecture, including queuing, delivering, storing, processing,
> and synchronizing events.

5.1.8 Implications. Given the wide range of safety and liveness challenges, we discuss the implications in two groups. The first covers event publishing and delivery semantics and the second relates to the challenges of synchronizing microservices' events.

CAS 1, 2, 4, and 7. Developers lack a comprehensive specification to guide them with the variety of event processing semantics and in dealing with failures in asynchronous, distributed microservices, suggesting they are in need for supporting tools. For messaging technology providers and framework developers, their documentation can emphasize the publishing and delivery semantics offered more clearly, including, but not limited to, the possible failure scenario microservice developers must consider and example workarounds to mitigate some of their effects in the microservice state. In addition, these should be provided in a language suitable for developers who are not familiar with distributed and asynchronous systems.

For database providers, explicit APIs for rolling back microservices' underlying databases to a state prior to processing a given event can help alleviate the burden of dealing with online failures in event-based workflows. Researchers can systematically characterize the guarantees provided by popular messaging technologies and enhance code analysis tools to better alert developers about potential hidden shortcomings of the different guarantees.

CAS 3, 5, and 6. The need to process and match distinct event streams generated by different producer microservices spans different functional requirements, such as replicating data, processing distinct but correlated events, and maintaining queries over event streams. Together, they form 26.83% of the total challenges, highlighting the practical significance of these event processing patterns. To fulfill these, developers often hold the assumption that events will arrive in a timely and exactly once manner. However, events may either take arbitrary time to arrive or never arrive. Besides, even if events arrive, they may be repeated. The mismatch between expectations and reality leads to frustrations in implementing the requirements above. Thus, microservice developers must be vigilant with designs that favor the existence of such event processing patterns. Rethinking service boundaries [1] may lead to some of the events involved in intricate event processing patterns being merged or eliminated.

Providers of messaging technologies and cloud computing services, and framework developers can offer further guidance at the documentation and API levels to alert developers regarding the potential dangers of some event processing patterns. Potentially, message technologies can devise better APIs to support developers making sure event dependencies are met. For example, an API to wait for multiple dependant events instead of individual events.

Researchers can improve static and dynamic analysis tools to detect certain code properties and execution traces that may lead to the harmful scenarios discussed. Another direction could be developing automatic tools to enforce safety properties on event processing. For example, Lesniak et al. [62] extend message queuing system with a programmable interface where developers can specify event processing order which will be enforced automatically.

All CAS. Researchers can devise or extend existing programming models to account for use cases that necessitate correlation between distinct event streams and abstract publishing and delivery semantics from the microservice code. For instance, they can take inspiration from the dataflow model [2], commonly used in stream processing engines, to accommodate the missing dynamic topology, non-blocking, isolated failure model found in real-world microservices. That can simplify the programmability of microservices and favor the adoption of tricky event stream processing patterns. Additionally, this programming model can unify the design and deployment of event-based microservices across cloud providers and execution platforms to facilitate the migration and hybrid execution of microservice applications across message technology providers, an emerging trend in cloud computing [25].

5.2 Event Schema Management

Microservice practitioners model events for external interaction. In other words, the decoupling nature of a microservice design requires producers to encode all the necessary data so that consumers do not need to query the producer for additional data when processing such an event. Furthermore, the events generated by microservices must comply with a data type to be correctly processed by consumers, and, as with any other data type, it can evolve over time. This section discusses the challenges associated with modeling and evolving event schemas in microservice architectures.

5.2.1 Modeling event schemas (CEM1). We observe that how events are modeled and what they represent semantically may dictate the design and performance of microservices. For example, a developer looks for recommendations for event design explaining the following scenario [23]:

Say you have a micro-service architecture where multiple services produce and consume *unit_statuses*.
There are multiple ways to design this.

The person continues explaining some options in mind:

1. Create a **generic topic `unit-status`** and make services consume and produce messages on this topic.
[...]
2. Create a **specific topic for each status**, for example `unit-status-created`,
`unit-status-packaged`, `unit-status-loaded`, `unit-status-deleted`, etc.

In sequence, the practitioner explains the drawbacks of each:

1. This has the consequence that **you consume your own messages and have to filter them**. 2. Requires
a **code or configuration change** in potentially all service when a new status topic is added.

In another question [123], a similar inquiry is presented where a practitioner wonders about two event design
approaches:

(1) *A* sends a message which contains event and related entity id like: `entityCreated { entityID: 1234 }`.
B consumes this message and **if it needs further information, it fetches this from A** with `entityID`
(2) The message not only contains the information above, but **also metadata** like: `entityCreated {`
`entityID: 1234, SomeFieldKey: someFieldValue, ... }`

The practitioner continues addressing the pros and cons of each and wonders which one to choose:

(1) Pros: **Less network usage; Always the same structure of messages**. Cons: If information from A
is needed on demand there **must be some mechanism to catch**, e.g. network failures.
(2) Pro: **Information is already there**. Con: What if the **attached information is not enough**?

Although practitioners seem aware of some of the consequences different event modeling options bring to the
application design, the excessive number of questions, 19.71% of the total, suggests developers encounter uncertainties
in deciding on an optimal event design.

Discussion: Differently from relational model [26], which provides a formal foundation that allows database designers
to reason about the possible performance implications of a data model (e.g., the level of normalization dictates whether
additional `JOIN` operations are necessary to retrieve certain information from the database), event design is still an
open problem. As a result, developers navigate through many possible event designs that, if not carefully thought out,
may impact the design of the application and the performance of event-triggered operations.

5.2.2 Evolving event schemas (CEM2). We also observe some microservice practitioners share uncertainties on how
to deal with the possible effects of event schema evolution properly. In particular, as events are often parsed into
microservices' own data models, practitioners express concerns over the impact of event schema changes in their
microservice states. For instance, upon a new attribute being incorporated into an event's schema (user's `address`), a
developer wonders how to update a microservice state retroactively [47]:

(DEV#1) I can update my event and **now include the `address`**, but it will only work for new users, **the
old ones will have *null* addresses**. Should I scan the whole database and manually dispatch an event
for each user?

On the other hand, although events are considered abstractions to decouple microservices, some practitioners report
event schema evolution impacting dependent microservices [87]:

(DEV#2) services are tightly coupled by [event] schema and **can causes errors and this contradicts with event sourcing goal**. How can we address this problem?

Evolving event schemas also appear as an important challenge, corresponding to 20.59% of the questions about managing event schemas in microservices and accounts for 5.02% of the total questions.

Discussion: We observe that developers are aware of the risks associated with schema changes and their potential impact on microservices dependent on the modified event type. If the type of incoming events does not match the expected event type in a consumer microservice, the processing of this event processing may fail. However, developers have not found a systematic solution to either minimize the impact of or comprehensively manage the evolution of event schemas across the network of dependent microservices.

State-of-the-practice data serialization formats like Apache Avro [12] can potentially ease this challenge. However, such tools require a global adoption across all microservices, forcing every event producer and consumer to employ schema and data contract enforcement. Besides, adopting such tools is not always possible because the messaging technology adopted may not provide native support. In overall, this remains an open challenge in practice.

Finding 5: Developers find no principled ways to model events accounting for performance and decoupling trade-offs. Systematic support for managing event schema changes is also an open problem.

5.2.3 Implications. With the lack of automatic support for event schema evolution, microservice developers can isolate failures caused by event schema mismatches and log them appropriately for later reconciliation. Messaging technology providers can supplement their documentation with more technical examples of how developers can isolate the microservices' application code from the impact of event schema changes. Further guidelines on appropriately rolling out event schema changes without impacting individual microservices in the context of their technologies can also benefit developers [78]

Researchers can develop analysis tools that holistically map the microservice event topology, matching producer and consumer microservices through the events exchanged. The tool can incorporate metrics to aid developers in reasoning about the trade-offs of different event designs. Furthermore, researchers can provide a formal foundation for modeling events in microservice applications. The model must allow for reasoning about the performance trade-offs of including specific attributes in events. Lastly, automatic support for event schema evolution across microservices is an open and complex problem that researchers could tackle.

5.3 Performance

Performance is systematically cited as an important factor for adopting microservice architectures [58, 70, 83]. Developers seek to reap the benefits of having the application decomposed into independently scalable building blocks [59]. In an architecture based on events, though, the performance of individual microservices is tightly coupled with its capacity to process and dispatch application-generated events at a rate that does not compromise the expected performance of the downstream microservices in the event flow. As a result, disruptions in the event flow can lead to critical performance impacts. In this section, we analyze such cases from the lens of microservice developers.

---

[12]https://avro.apache.org

5.3.1 Event-processing overhead (CP1). Developers report particular event processing scenarios that ultimately lead to a performance penalty. For example, in connection with **CAS5**, some practitioners report the need to replay events to fulfill a given requirement, but that ends up introducing additional load to the application [22]:

> The validation of a **reservation request needs to know the previous and following reservations** (within 3 hours from the booking time of the incoming request)

> **if I ask for a reservation for tomorrow, the system will replay reservations from 6 months ago** that usually are not related with the incoming request

The developer then highlight the overhead:

> This leads to inefficiencies over time as result of the **huge amount of unnecessary events that are replayed**. I thought to solve it using daily snapshots but it seems the wrong way to do it.

In a similar way, in connection with CAS3, other developers acknowledge there could be a performance penalty on leveraging events for replicating data across microservices. For example [65]:

> The *Invoices* microservice must listen to all `ContactCreated` and `ContactDeleted` events in order to know if the given recipient id is valid.

> Then **I'd have thousands of Contacts** within the *Invoices* microservice, **even if I know that only a few of them will ever receive an `Invoice`.** Is there any best practice to handle those scenarios?

Practitioners express uncertainties in dealing with the overhead of processing events and their possible performance impacts. This challenge is the most prominent in the `Performance` category (53.57%), and spans 5.38% of the total questions. This suggests that microservices may execute with suboptimal performance in real-world deployments. **Discussion:** At first sight, the overhead of the first case can be potentially mitigated by storing the events processed in the microservice's private state. However, this technique introduces hidden dangers. For example, keeping track of the last stored event is nontrivial and can ultimately lead to losing an event due to crashes. Besides, developers tend to favor the data shipping paradigm [14]. By offloading state management to another system (e.g., database system, message broker, or cloud object storage), applications can remain stateless, decreasing their complexity. In the second case, the overhead is triggered by the lack of systematic support for data replication through event management.

5.3.2 Large event payload (CP2). We observe developers also present concerns over the size of the event payload generated and consumed by different microservices. For instance, a common concern lies in whether publishing events with large payloads into the message layer entails a good practice:

> (DEV #1) My question is, is it wise to stream such file over event bus from Service A to API gateway? **(File may get as large as 100 MB**) [90]

> (DEV #2) If an event needs to pass a **very large volume of data** to the next Saga event, how is this done in terms of the request structure? Is it divided into multiple Sagas for example (as a result pagination type)? [42]

> (DEV #3) I think we will need to go with second option i.e. not sending data in the event due to the **potential size of the event data**. [95]

To circumvent the potential performance degradation incurred by processing large event payloads, some developers look for alternatives to the solo event-driven approach but end up encountering other issues:

(DEV #3) I was also thinking about adding the URI in the event so that the service just needs to call it. Only issue I can see is that **how would the service know what type to deserialize** the response to? [95]

The performance impacts of generating, processing, and storing large event payloads account for 17.86% of `Performance` challenges. This challenge represents 1.79% of the total questions and highlights that developers find difficulties on dealing with large event payloads and their consequences to microservice performance.

Discussion: Message brokers and modern log processing systems are not designed to handle messages with large payloads natively. Message brokers have historically targeted systems' integration cases, which usually do not require large event payloads [83]. Log processing systems, like Kafka, for example, were initially designed for the timely processing of logs extracted from processes running in distributed servers [57]. Similarly to message brokers, though, it was never a design goal to accommodate the processing of large log payloads. Native support for these types of objects is found in cloud storage services and specific column-oriented database systems, like `BLOB` type in PostgreSQL. [13] Therefore, the questions analyzed in StackOverflow suggest that microservice developers may benefit from either redesigning their computations to manage large objects in appropriate storage systems or rethinking the granularity of their event payloads when they grow arbitrarily. This may involve breaking down application components into smaller, finer-grained tasks to decrease the size of events transmitted across microservices.

5.3.3 Fluctuating event rate (CP3). Developers describe challenges associated with dynamic workloads and their impact on the observed event processing rate.

(DEV #1) In case of **pressure over the system**, can service B communicate to service A to **slow down**, and A will react to this by **not accepting more requests** from clients, till B decides it can continue? Is this something that can be achievable using [(omitted library)]? [86]

(DEV #2) I want to **restrict the users from publishing a new message** on this topic to **prevent** my system from **choking after a certain limit**.

For example, if the number of unacknowledged messages in the topic is already **more than or equal to 10000**, then I want to give a bad input **exception** or something to restrict users from flooding my queue. [32]

(DEV #3) **Traffic is increasing** and we've noticed that events **spend a lot of time in queue**. We need to **process events faster**. [...] Is it possible to implement circuit breaker or a failover mechanism for an async call? [54]

The challenges about adapting microservices to handle increased influx of events represent 28.57% of `Performance` questions. Representing 2.87% of the total of questions, this additional performance challenge strengthens the perception that microservice developers encounter difficulties on reaching performance goals in event-driven microservice architectures.

Discussion: With traditional synchronous communication paradigms, like RPC and HTTP requests, an application can usually define a threshold of the maximum allowed concurrent connections. In event-based architectures, though, there is an indirection: the events are often stored in the event management layer first and only later forwarded to (or pulled by, depending on the system) consumers. This decoupling introduces challenges for consumer microservices to communicate whether producers must decrease the event generation rate or even refrain from generating new events. On the other hand, modern event management systems, like Kafka, provide abstractions to parallelize the

---

[13] https://www.postgresql.org/docs/16/largeobjects.html

consumption of events through partitions. However, developers must configure the unit of parallelism according to their requirements, which may not be trivial for newcomers.

> **Finding 6:** Developers encounter challenges on managing the event rate across producers and consumers. This only exacerbates in the presence of event replays and large event payload sizes.

5.3.4 Implications. Message technology vendors can provide efficient and proactive event-processing techniques and algorithms that mitigate or decrease the impact of fast producers and slow consumers. As mentioned in the discussion of individual challenges, developers may benefit from rethinking the granularity of their microservices and the content included in their event payloads to escape from adding overhead to their event processing pipelines. Developers may also be aware that, as events flow through microservices, the events generated by each microservice may include additional data. That suggests developers may track the event payload sizes more carefully as they flow across possibly many microservices.

Researchers can investigate whether state-of-the-art approaches for scaling distributed systems also apply to event management in microservices. Besides, researchers can develop systematic tools to mitigate the effects of large payload sizes in event-based microservice workflows. That could involve optimal deserialization of event payloads (e.g., only deserializing the subset of attributes that the microservice requires), advanced compression tools that mitigate the overhead of growing events, and transparent removal of unused event attributes.

5.4 Observability

Observability is a critical concern in modern software development because it allows developers to track important application behavior and metrics, such as application exceptions and memory usage, respectively [11]. In distributed systems, such as microservice architectures, observability becomes more critical since problems can possibly span across multiple components that execute independently in a decentralized manner [11, 113].

In our analysis, we were surprised to find that microservice developers leverage events not only as an abstraction to process operations asynchronously but also to track the progress of operations across multiple microservices and troubleshoot the complex interplay of multiple microservices. In this section, we describe these patterns and their associated challenges.

5.4.1 Event flow observability (CO1). Although tools to observe synchronous requests (HTTP or RPC-based) crossing microservices enjoy apparent consolidation, developers express challenges in monitoring the asynchronous event flow across microservices, as exemplified by the two following developers' quotes:

> (DEV #1) What is something we can do to have an **end to end understanding** from producers to consumers, etc? Mainly for troubleshooting purposes/change management. [31]

> (DEV #2) What is the best way **to track the JSON as it flows** through many microservices down stream in an event driven way? [96]

Another example presents a developer wondering whether an observability tool can embrace events:

> (DEV #3) Will [(omitted tool)] **be able to trace requests done over the eventbus?** The tracing page says that headers need to be propagated through in http or grpc - but the eventbus sends messages via tcp – does that mean that [(omitted tool)] will not be able to trace requests and show the visualisation tools[?] [7]

Furthermore, we found that issues in the message broker only exacerbate the challenge of monitoring how healthy the event flow is, as explained by a developer [105]:

> For any reason **the messages get stuck after some time [...] There is nothing in the log files nor the OS event log.** I have to restart the ([omitted technology]) service in order to "reanimate" it.
>
> Afterwards[,] all stuck messages will be processed and **everything is working fine until the next "accident".** [...] Does anybody has an idea what I could check additionally to find out what the problem is?

Observing the event flow appear significantly in *Observability* challenges (89.47%). This highlights the difficulties developers find to understand the progress of events as they flow across microservices.

Discussion: Application-generated events serve as natural progress markers. It is a natural choice for developers to obtain an end-to-end overview of the application execution based on these events. However, developers encounter a twofold problem. On the one hand, developers find that state-of-the-art observability tools do not natively support tracking application-generated events. On the other hand, when disruptions in the event flow are not caused by crashes or bugs in the microservices themselves, developers have difficulties in finding the root problem. Although messaging technologies are only part of the intricate and heterogeneous components in microservice deployments [59], they are a core enabler of event-based interactions in microservices. Thus, timely detection of message broker failures and their relationships with microservice disruptions are key to observability in event-based architectures.

5.4.2 Auditing via event replay (CO2). In CAS5, we discussed how developers leverage events in an attempt to synchronize data across microservices. As a related pattern, we also found that developers use the events as an abstraction to reproduce some application behavior. Similar to CAS5, developers end up facing difficulties in ensuring the correctness of the process, as explained by a developer [107]:

> (DEV#1:) If the VisitId was deleted across all services we could just **replay the events** one at a time, in order, and reproduce an exact copy of the original record.

The developer then mentions an impedance commonly expressed by another practitioner across the questions:

> I've been using [(omitted message technology)] for the stream itself [...] The issue though is that **this is not suitable for a replay store** - only delivery of the event messages.

Another developer warns about the hidden dangers of such a method [116]:

> (DEV#2:) What's maybe not a good idea is trying to **recover the current state** of your domain model by replaying an arbitrary set of your events.
>
> Remember, **an event isn't usually a complete representation of the state of the model after the change**, but rather a description of the things that changed. (..)

Auditing past microservice executions through events to observe and further understand their behavior appear in 10.53% of `Observability` challenges. This suggests that current technologies offer insufficient abstractions to support event replay effectively.

Discussion: As events often represent intermediate microservice states, they appear as convenient abstractions to developers seeking to troubleshoot past application behavior. However, in the same vein as CAS5, this practice incurs hidden challenges due to the lack of appropriate support from current messaging and framework technologies. For instance, as discussed in CAS7, mismanaging the processing of duplicate events and their possible effects on the

application can lead to corrupting a microservice state. Another example, based on CEM1, is that microservices may even fail to replay events due to the divergence of event schemas.

> **Finding 7:** Developers find difficulties on observing and thus reacting upon disruptions that occur in the event flow across microservices. These disruptions may not only be caused by microservices, but also message brokers and network partitions.

5.4.3 Implications. Messaging technology providers can evolve their systems and APIs to provide native integration with industry-strength observability tools. The integration must account for the needs of developers in their real-world cases as discussed above, including but not limited to specific metrics such as event processing and acknowledgment delays from consumers.

For developers, as state-of-the-art abstractions render limited support for safely replaying events, developers can use a "staging" environment (similar to testing environments with testing, staging, and production [110]) to audit their event-triggered operations. However, generating an application and message broker state from a previous point in time for replaying can be challenging.

In the tool development landscape, opportunities for researchers are: (i) They can co-design static analysis tools and dynamic analysis of event streams for a holistic monitoring of microservice event streams. In particular, this can aid developers by automatically matching correlated events and associated service disruptions in the event flow. (ii) They can develop techniques for isolating re-execution of events from impacting the actual microservice states. Recent tools [63] allows for replaying operations in database-backed applications. However, it is unclear how the approach can be applied to events and distributed states of microservices.

5.5 Security

Security is another critical concern in event management in microservices. In this section, we discuss developers' requirements and challenges when ensuring that event ingestion, processing, and storage meet security constraints.

5.5.1 Authentication (CS1). Although events are often an abstraction that is only internally recognized by the microservices, some events may be generated by end users. That requires making public APIs available, which necessarily exposes the microservice to security vulnerabilities, as exemplified as follows:

> (DEV#1) Because I **don't check authorization** on websocket open, in theory this approach is vulnerable to a **dDos attack**, where an attacker simply opens as many sockets as they can. [18]

On the other hand, even when APIs are safeguarded with authentication methods, developers are concerned about aligning the validity of the authentication mechanism with the event flow.

> (DEV#2) In case of event driven application, it's possible **to ensure the token is valid?** In case of failure, the user clicks on button and an event is written but the processing of this will be hours later. How can I process the event with the user credentials? [66]

> (DEV#3) if I change my communication mechanism to using **async messaging** (e.g. RabbitMQ), how would I now authenticate the event to the scope of the user who initiated the event. [108]

Challenges on ensuring event-based workflows and event-triggered computations execute securely appear substantially in `Security` challenges (76.92%). Spanning across 3.58% of the total questions, that suggests developers fall short on a holistic solution to secure their event-driven microservices.

Manuscript submitted to ACM

Discussion: Aligning authorization mechanisms and event processing is non-trivial. One obstacle is that the authorization and messaging mechanisms are often provided by different systems, resulting in a system integration issue. In particular, developers working with microservices need to handle authentication token life cycles and make sure that any events related to the token are in line with its validity. This process can add complexity to the solution.

5.5.2 Data Privacy (CS2). Data privacy emerged as a key requirement in modern applications to prevent leakage of user data and potential violation of data privacy laws such as the General Data Protection Regulation (GDPR) [14]. Data privacy laws prescribe that users can request the removal of their data at any moment, and firms are usually given a deadline for fulfilling the request. As discussed earlier, in microservices, data flows from the multiple event streams to many microservice consumers, which in turn apply operations on their private states. This heterogeneous data placement and movement can make managing data privacy challenging. As exemplified by the following quote [64], we observe that microservice developers express uncertainties about handling data privacy in event management.

> **deleting user data** in an event-sourced system [...] **To which stream is the tombstone event written?** The event stream of the specific user? Or is there an event stream specifically for tombstone events?

> **to keep some data** (e.g. the users' id), he advises to split the user stream into a **public and private event stream**. [...] How can you make sure that both events have been appended?

Managing data privacy in the context of event streams accounts for 23.08% of security challenges. This suggests that developers find uncertainties on how to properly manage data privacy in events exchanged between microservices.
Discussion: We find that developers often manage data privacy in event streams using ad-hoc solutions, such as separating privacy and non-privacy data into different streams. However, these mechanisms do not enforce by design that data leakages do not occur, such as user data contained in events that are not supposed to be processed by certain consumer microservices. Besides, these mechanisms can lead to consistency problems when it is up to the developer to ensure privacy and non-privacy streams are in sync.

Finding 8: It is unclear for microservice developers how to properly safeguard security properties in the context of event processing in microservices.

5.5.3 Implications. Messaging technology providers could provide technology-specific guidelines on properly handling authentication and data privacy in their systems, accounting for the mechanisms that prevent exposing unintended event payloads. Cloud providers and framework developers can offer custom deployment templates to facilitate meeting certain security criteria. However, these should not be oblivious to the event management layer, suggesting that an appropriate cross-system integration is necessary to fulfill security challenges.

Microservice developers can, by principle, never include sensitive data in their generated events (i.e., privacy-by-design). However, this can jeopardize the decoupling benefit brought by event-based architecture, requiring consumers to contact the producer for the missing sensitive data. Researchers can extend existing code analysis tools to alert developers about bugs and configuration mistakes that can lead to unintended access to events and exposure of private data in event streams. Besides, researchers can investigate privacy-by-design event management methods without impact decoupling. For instance, protocols that automatically encrypt sensitive data in events without developer intervention.

---

[14]https://gdpr.eu/

Summary: Microservice developers encounter a myriad of problems when managing events, including dealing with large event payloads, evolving event schemas, auditing and validating security tokens through events, and processing events in order.

6 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our empirical study.

Selection bias of the source. Similar to previous research [8, 121, 124], our work uses SO as the data source to study the challenges developers encounter in practice. Although other data sources could supplement the findings, the heterogeneity of challenges, application scenarios, and developer expertise found in SO enables a trustworthy reflection of the state of the practice. For instance, the findings are aligned with several blog posts that report similar challenges [20, 35, 44, 75, 76, 97, 98, 101, 102, 125].

Construction of tag set. We select a set of tags to filter SO questions associated with event management in microservice architectures. The relevant keywords and metric applied may omit some tags associated with event management in microservices. To mitigate this threat, we rely on manual inspection conducted by independent researchers, and we adopt the lowest threshold used in previous work [8, 121, 124] to include the highest amount of tags as possible.

Subjective of researchers. Since we adopt manual analysis to identify and categorize challenges, that may threaten our findings' validity. To minimize this threat, the first two authors analyzed and classified each filtered question in isolation. Upon conflicts, an experienced arbitrator coordinated the discussion to reach an agreement.

Generalizability of our findings. We recognize that the application scenarios, requirements, design choices, event processing patterns, and the challenges extracted from SO may not be generalizable to all possible cloud and data platforms to which event-driven microservices can be deployed to. However, as SO is an industry-strength questions & answers platform for developers, their questions tend to shed light on the most adopted industrial practices. Thus, the 10-year question range covered in this study strengthens our perception that the findings highlight a substantial portion of the challenges in the state of the practice.

7 RELATED WORK

In this section, we summarize related work on challenges that developers face in microservice architectures.

Benefits of microservice adoption. Zhang et al. [127, 128] conduct interviews with microservice practitioners to understand the benefits brought about by microservice adoption. They find organizational transformation, decomposition, distributed monitoring, and bug localization as prominent challenges. Wang et al. [118] interview and survey practitioners to collect and categorize best practices, challenges, and their related successful solutions employed by practitioners. They find managing API changes, lack of support for monitoring, and finding microservice granularity as common challenges.

Microservice issues. Zhou et al. [129] performed a survey to characterize typical faults, debugging practices, and the challenges entailed by troubleshooting failures in microservice architectures. They find microservice developers can benefit from improved trace visualization tools, specially in cases related to microservice interactions. Ramírez et al. [89] mine SO posts to identify common issues on development and testing microservice applications. They find missing parameters as a prominent issue in communicating with other microservices, wrong library versions as the main impediment for service discovery, and connecting to other microservices with correct user credentials as an example of authentication & authorization challenge. Waseem et al. [119] employ a mixed-method empirical study to understand

the types of issues microservice developers experience. Similarly to Ramírez et al. [89], they find programming errors, missing artifacts, invalid configuration and communication as the main causes behind the issues.

Data Management in Microservices. Laigner et al. [59] characterize data management challenges in microservice architectures through a mixed-method empirical study. They focus on analyzing developers' pitfalls when implementing data management logic in the application layer and discussing the limitations of state-of-the-art database systems that prevent them from better serving microservice architectures.

Event Sourcing Pattern. Overeem et al. [78] study the challenges practitioners experience using event sourcing. Through interviews with 25 engineers, they find event system evolution, steep learning curve, lack of available technology, rebuilding projections, and data privacy as the main issues affecting the development of systems that employ the event sourcing pattern. From 19 systems mentioned by interviewees, only 8 apply the microservice architectural style.

Security in Microservices. Nasab et al. [71] performs a mixed-method empirical study to understand the security practices in microservice systems. They collect 28 security practices open-source repositories and SO posts, which were later confirmed through a survey with practitioners.

Exploratory Studies in Microservices. Hacaloglu and Demirors [46] study the usefulness of events for software size measurement. They find preliminary evidence that events can supplement software size measurement techniques. Lazzari and Farias [36] reports an exploratory study that compares event-driven and REST architectural styles in the context of modularity. They find preliminary evidence that event-driven architecture improves the separation of concerns.

Overall, we observe that challenges related to event management are overlooked in the literature. For instance, although few works report developers mentioning possible issues with message technology systems [71, 119] and asynchronous task invocation [89, 129], they are often addressed as general microservice configuration or invocation problems, preventing a proper characterization of the problem. Therefore, although event management in microservices has been rapidly gaining industry popularity [20, 35, 44, 59, 75, 76, 97, 98, 101, 102, 125], it has not received due attention from the research community. In this work, we make the first attempt to investigate the specific challenges that developers face when managing events in microservice architectures.

8 CONCLUSION

In this paper, we mine and analyze several relevant Stack Overflow questions to characterize the state of the practice and the challenges microservice developers face while managing events. We identify key patterns developers use while attempting to realize their functional and non-functional requirements, suggesting a tension between achieving requirements related to data consistency, loose coupling, and performance.

To further understand these tensions, we manually examined 628 sampled questions and identified key challenges microservice developers face in varied application scenarios. These include issues related to queuing, delivering, and processing events at the application level, as well as monitoring and securing the event flow. Based on the myriad of practical findings, we provide actionable implications for messaging systems, framework maintainers, cloud providers, and researchers.

We hope that the results drive the reflection of microservice developers and researchers to escape from the dangers of asynchronous, event-based designs and to build event management technologies that meet the expectations of microservice developers respectively.

ACKNOWLEDGMENTS

REFERENCES

[1] Yalemisew Abgaz, Andrew McCarren, Peter Elger, David Solan, Neil Lapuz, Marin Bivol, Glenn Jackson, Murat Yilmaz, Jim Buckley, and Paul Clarke. 2023. Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. *IEEE Transactions on Software Engineering* 49, 8 (2023), 4213–4242. https://doi.org/10.1109/TSE.2023.3287297

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment* 8 (2015), 1792–1803.

[3] annemartijn. [n. d.]. *How to recover from missed integration or notification events in event driven architecture?* Retrieved July 29, 2024 from https://stackoverflow.com/questions/65425071/how-to-recover-from-missed-integration-or-notification-events-in-event-driven-ar

[4] .NET Application Architecture Reference Apps. 2024. eShopOnContainers. https://github.com/dotnet-architecture/eShopOnContainers

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A view of cloud computing. *Commun. ACM* 53, 4 (apr 2010), 50–58. https://doi.org/10.1145/1721654.1721672

[6] Imran Arshad. [n. d.]. *Microservices client acknowledgement and Event Sourcing.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/54451013/microservices-client-acknowledgement-and-event-sourcing

[7] Asad Awadia. [n. d.]. *Istio request tracing for vert.x event bus messages.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/51123104/istio-request-tracing-for-vert-x-event-bus-messages

[8] Mehdi Bagherzadeh and Raffi Khatchadourian. 2019. Going Big: A Large-Scale Study on What Big Data Developers Ask. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 432–442. https://doi.org/10.1145/3338906.3338939

[9] Alan Bandeira, Carlos Alberto Medeiros, Matheus Paixao, and Paulo Henrique Maia. 2019. We Need to Talk About Microservices: an Analysis from the Discussions on StackOverflow. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, Montreal, QC, Canada, 255–259. https://doi.org/10.1109/MSR.2019.00051

[10] Tim Berglund. [n. d.]. *Introduction to Apache Kafka Partitions.* Retrieved June 8, 2024 from https://developer.confluent.io/courses/apache-kafka/partitions/

[11] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2020. Visualizing Distributed System Executions. *ACM Trans. Softw. Eng. Methodol.* 29, 2, Article 9 (mar 2020), 38 pages. https://doi.org/10.1145/3375633

[12] Michal Borowiecki. [n. d.]. *Data Consistency Across Microservices.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/43950808/data-consistency-across-microservices/44748028#44748028

[13] Susanne Braun, Stefan Deßloch, Eberhard Wolff, Frank Elberzhager, and Andreas Jedlitschka. 2021. Tackling Consistency-related Design Challenges of Distributed Data-Intensive Systems: An Action Research Study. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '21)*. Association for Computing Machinery, New York, NY, USA, Article 20, 11 pages. https://doi.org/10.1145/3475716.3475771

[14] Sergey Bykov, Alan Geller, Gabriel Kliot, Jim Larus, Ravi Pandya, and Jorgen Thelin. 2010. *Orleans: A Framework for Cloud Computing.* Technical Report MSR-TR-2010-159. https://www.microsoft.com/en-us/research/publication/orleans-a-framework-for-cloud-computing/

[15] Hebert Cabane and Kleinner Farias. 2024. On the impact of event-driven architecture on performance: An exploratory study. *Future Generation Computer Systems* 153 (2024), 52–69.

[16] CallMeTheBreeze. [n. d.]. *What is the best way to implement a fast, scalable statistics aggregation architecture?* Retrieved July 29, 2024 from https://stackoverflow.com/questions/30583466/what-is-the-best-way-to-implement-a-fast-scalable-statistics-aggregation-archit

[17] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (2017), 1718–1729.

[18] John Chang. [n. d.]. *Can I rely on ConnectionId for security with API Gateway Websockets?* Retrieved July 29, 2024 from https://stackoverflow.com/questions/61737655/can-i-rely-on-connectionid-for-security-with-api-gateway-websockets

[19] Chunyang Chen and Zhenchang Xing. 2016. Mining technology landscape from stack overflow. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[20] Dmitry Chornyi. [n. d.]. *Engineering Uber's Next-Gen Payments Platform.* Uber Technologies Inc. https://eng.uber.com/payments-platform (Accessed on 2021-03-08).

[21] Christian Paesante Chris. [n. d.]. *CQRS - out of order messages.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/53270770/cqrs-out-of-order-messages

[22] Christian. [n. d.]. *Event sourcing - Event streams clarification*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/53949531/event-sourcing-event-streams-clarification

[23] Christophe. [n. d.]. *Is it bad practice to produce and consume messages from the same topic?* Retrieved July 29, 2024 from https://stackoverflow.com/questions/45486658/is-it-bad-practice-to-produce-and-consume-messages-from-the-same-topic

[24] L. Chung, B.A. Nixon, E. Yu, and J. Mylopoulos. 2012. *Non-Functional Requirements in Software Engineering*. Springer US, New York, NY, USA. https://books.google.dk/books?id=MNrcBwAAQBAJ

[25] Google Cloud. [n. d.]. *What is a Hybrid Cloud?* Retrieved June 19, 2024 from https://cloud.google.com/learn/what-is-hybrid-cloud

[26] E. F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (jun 1970), 377–387. https://doi.org/10.1145/362384.362685

[27] code pendent. [n. d.]. *How to implement a microservice Event Driven architecture with Spring Cloud Stream Kafka and Database per service*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/42140285/how-to-implement-a-microservice-event-driven-architecture-with-spring-cloud-stre

[28] codependent. [n. d.]. *Spring Cloud Stream Kafka - Eventual consistency - Does Kafka auto retry unacknowledged messages (when using autocommitoffset=false)*. Retrieved June 19, 2024 from https://stackoverflow.com/questions/42230797/spring-cloud-stream-kafka-eventual-consistency-does-kafka-auto-retry-unackno

[29] code_pendent. [n. d.]. *Spring Cloud Stream Kafka - Eventual consistency - Does Kafka auto retry unacknowledged messages (when using autocommitoffset=false)*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/42230797/spring-cloud-stream-kafka-eventual-consistency-does-kafka-auto-retry-unackno

[30] Dapr. [n. d.]. *Workflow patterns*. Retrieved July 10, 2024 from https://docs.dapr.io/developing-applications/building-blocks/workflow/workflow-patterns

[31] deblearns1. [n. d.]. *Kafka implementation - how to see end to end workflows with microservices*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/65500926/kafka-implementation-how-to-see-end-to-end-workflows-with-microservices

[32] Shalaka Deshpande. [n. d.]. *How to restrict publishing to a topic in GCP pub/sub based on number of unacknowledged messages in the queue?* Retrieved July 29, 2024 from https://stackoverflow.com/questions/74064658/how-to-restrict-publishing-to-a-topic-in-gcp-pub-sub-based-on-number-of-unacknow

[33] Santanu Dey. [n. d.]. *Data Consistency Across Microservices*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/43950808/data-consistency-across-microservices

[34] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. 2018. Migrating Towards Microservice Architectures: An Industrial Survey. In *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, Seattle, WA, USA, 29–2909. https://doi.org/10.1109/ICSA.2018.00012

[35] Uber Engineering. 2020. *Revolutionizing Money Movements at Scale with Strong Data Consistency*. Uber Technologies Inc. https://eng.uber.com/money-scale-strong-data (Accessed on 2021-03-08).

[36] Kleinner Farias and Luan Lazzari. 2023. Event-driven Architecture and REST Architectural Style: An Exploratory Study on Modularity. *Journal of Applied Research and Technology* 21, 3 (Jun. 2023), 338–351. https://doi.org/10.22201/icat.24486736e.2023.21.3.1764

[37] Danica Fine and Nikoleta Verbeck. 2022. *Diagnose and Debug Apache Kafka Issues: Understanding Increased Consumer Rebalance Time: Increased Consumer Rebalance Time*. Retrieved June 19, 2024 from https://www.confluent.io/blog/debug-apache-kafka-pt-3/

[38] Martin Fowler. 2006. *Focusing on Events*. Martin Fowler. Retrieved July 3, 2024 from https://martinfowler.com/eaaDev/EventNarrative.html

[39] Martin Fowler. 2017. *What do you mean by "Event-Driven"?* Martin Fowler. Retrieved June 22, 2024 from https://martinfowler.com/articles/201701-event-driven.html

[40] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2023. A Survey on the Evolution of Stream Processing Systems. arXiv:2008.00842 [cs.DC] https://arxiv.org/abs/2008.00842

[41] Paolo Francesco, Ivano Malavolta, and Patricia Lago. 2017. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In *International Conference on Software Architecture*. 21–30. https://doi.org/10.1109/ICSA.2017.24

[42] Victor França. [n. d.]. *Saga Choreography implementation problems*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/48487098/saga-choreography-implementation-problems

[43] Renato Gama. [n. d.]. *How to trigger an action only after receiving two or more events in an event driven architecture?* Retrieved July 29, 2024 from https://stackoverflow.com/questions/51421205/how-to-trigger-an-action-only-after-receiving-two-or-more-events-in-an-event-dri

[44] Adam Gluck. [n. d.]. *Introducing Domain-Oriented Microservice Architecture*. Uber Technologies Inc. https://www.uber.com/en-DK/blog/microservice-architecture/

[45] Nicolas E Gold and Jens Krinke. 2022. Ethics in the mining of software repositories. *Empirical Software Engineering* 27, 1 (2022), 17.

[46] Tuna Hacaloglu and Onur Demirors. 2023. An exploratory case study using events as a software size measure. *Information Technology and Management* 24 (04 2023), 1–20. https://doi.org/10.1007/s10799-023-00394-y

[47] hendoe. [n. d.]. *Approaches to update microservices databases retroactively*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/59923240/approaches-to-update-microservices-databases-retroactively

[48] Fabian Hueske and Vasiliki Kalavri. 2019. *Stream processing with Apache Flink: fundamentals, implementation, and operation of streaming applications*. O'Reilly Media.

[49] ignacy. [n. d.]. *Starting new Kafka Streams microservice, when there is data retention period on input topics.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/67074772/starting-new-kafka-streams-microservice-when-there-is-data-retention-period-on

[50] Workflow Patterns Initiative. [n. d.]. *Workflow Patterns.* Retrieved July 10, 2024 from http://www.workflowpatterns.com

[51] joe gates. [n. d.]. *how to deal with replication lag in microservices.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/71971389/how-to-deal-with-replication-lag-in-microservices

[52] Jordi. [n. d.]. *saga pattern: what about if compensation action fails.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/73108942/saga-pattern-what-about-if-compensation-action-fails

[53] jsc31994. [n. d.]. *Why is Kafka Streams in Node sending me duplicate payloads after I terminate the process and run once more?* Retrieved July 29, 2024 from https://stackoverflow.com/questions/63391504/why-is-kafka-streams-in-node-sending-me-duplicate-payloads-after-i-terminate-the

[54] Julio. [n. d.]. *Microservices async communication circuit breaker.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/68307199/microservices-async-communication-circuit-breaker

[55] KitKarson. [n. d.]. *Data Dependency Among Microservices.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/70509292/data-dependency-among-microservices

[56] Martin Kleppmann, Alastair R. Beresford, and Boerge Svingen. 2019. Online Event Processing: Achieving Consistency Where Distributed Transactions Have Failed. *Queue* 17, 1 (feb 2019), 116–136. https://doi.org/10.1145/3317287.3321612

[57] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. 1–7.

[58] Rodrigo Laigner, Marcos Kalinowski, Pedro Diniz, Leonardo Barros, Carlos Cassino, Melissa Lemos, Darlan Arruda, Sérgio Lifschitz, and Yongluan Zhou. 2020. From a Monolithic Big Data System to a Microservices Event-Driven Architecture. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* IEEE, Portoroz, Slovenia, 213–220. https://doi.org/10.1109/SEAA51224.2020.00045

[59] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *Proc. VLDB Endow.* 14, 13 (2021), XXX–XXX. https://doi.org/10.14778/3484224.3484232

[60] L. Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering* SE-3, 2 (1977), 125–143. https://doi.org/10.1109/TSE.1977.229904

[61] Lerman. [n. d.]. *nternal k8s services communication not balanced.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/73349618/internal-k8s-services-communication-not-balanced

[62] Anna Lesniak, Rodrigo Laigner, and Yongluan Zhou. 2021. Enforcing Consistency in Microservice Architectures through Event-Based Constraints. In *International Conference on Distributed and Event-Based Systems (DEBS).* 180–183.

[63] Qian Li, Peter Kraft, Michael Cafarella, Çağatay Demiralp, Goetz Graefe, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. 2023. R3: Record-Replay-Retroaction for Database-Backed Applications. *Proc. VLDB Endow.* 16, 11 (jul 2023), 3085–3097. https://doi.org/10.14778/3611479.3611510

[64] Florian Ludewig. [n. d.]. *Deleting Events with Tombstones and Public / Private Data in an Event-Sourced System.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/57784098/deleting-events-with-tombstones-and-public-private-data-in-an-event-sourced-sy

[65] Benjamin M. [n. d.]. *CQRS + Microservices: How to handle relations / validation?* Retrieved July 29, 2024 from https://stackoverflow.com/questions/42528718/cqrs-microservices-how-to-handle-relations-validation

[66] Matthias. [n. d.]. *Spring Security OAuth2 AuthorizationServer.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/29148547/spring-security-oauth2-authorizationserver

[67] Nabor C Mendonça, Craig Box, Costin Manolache, and Louis Ryan. 2021. The monolith strikes back: Why istio migrated from microservices to a monolithic architecture. *IEEE software* 38, 5 (2021), 17–22.

[68] Hamdy Michael Ayas, Philipp Leitner, and Regina Hebig. 2023. An empirical study of the systemic and technical migration towards microservices. *Empirical Software Engineering* 28, 4 (2023), 85.

[69] C. Mohan and B. Lindsay. 1985. Efficient commit protocols for the tree of processes model of distributed transactions. *SIGOPS Oper. Syst. Rev.* 19, 2 (apr 1985), 40–52. https://doi.org/10.1145/850770.850772

[70] Gastón Márquez, Mónica M. Villegas, and Hernán Astudillo. 2018. An Empirical Study of Scalability Frameworks in Open Source Microservices-based Systems. In *2018 37th International Conference of the Chilean Computer Science Society (SCCC).* 1–8. https://doi.org/10.1109/SCCC.2018.8705256

[71] Ali Rezaei Nasab, Mojtaba Shahin, Seyed Ali Hoseyni Raviz, Peng Liang, Amir Mashmool, and Valentina Lenarduzzi. 2023. An empirical study of security practices for microservices systems. *Journal of Systems and Software* 198 (2023), 111563.

[72] Arthur Navarro, Julien Ponge, Frédéric Le Mouël, and Clément Escoffier. 2023. Considerations for integrating virtual threads in a Java framework: a Quarkus example in a resource-constrained environment. In *DEBS'2023 - 17TH ACM International Conference on Distributed and Event-Based Systems*, ACM (Ed.). University of Neuchâtel, Neuchatel, Switzerland. https://doi.org/10.1145/3583678.3596895

[73] Sam Newman. 2015. *Building Microservices* (1st ed.). O'Reilly Media, Inc., Sebastopol, CA.

[74] nik2o. [n. d.]. *Join data in CQRS patterns from different microservices.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/57328269/join-data-in-cqrs-patterns-from-different-microservices

[75] Nubank. 2017. *Architecting a Modern Financial Institution.* Nubank. https://www.infoq.com/presentations/nubank-architecture/ (Accessed on 2022-02-11).

[76] Nubank. 2019. *Microservices at Nubank, an overview.* Nubank. https://building.nubank.com.br/microservices-at-nubank-an-overview (Accessed on 2021-03-08).

[77] Odsh. [n. d.]. *CQRS: project out-of-order notifications in an ElasticSearch read model.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/47516458/cqrs-project-out-of-order-notifications-in-an-elasticsearch-read-model

[78] Michiel Overeem, Marten Spoor, Slinger Jansen, and Sjaak Brinkkemper. 2021. An empirical characterization of event sourced systems and their schema evolution — Lessons from industry. *Journal of Systems and Software* 178 (2021), 110970. https://doi.org/10.1016/j.jss.2021.110970

[79] oznomal. [n. d.]. *How to monitor REST Endpoint for long running jobs.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/56923600/how-to-monitor-rest-endpoint-for-long-running-jobs

[80] Christian Paesante. [n. d.]. *AWS Event-Sourcing implementation.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/52632129/aws-event-sourcing-implementation

[81] Igor Petrov. [n. d.]. *SAGA and local transactions with Kafka and Postgres in Spring Boot.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/62223553/saga-and-local-transactions-with-kafka-and-postgres-in-spring-boot

[82] Praveen. [n. d.]. *Aggregates in Event Sourcing Pattern.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/49985156/aggregates-in-event-sourcing-pattern

[83] Dan Pritchett. 2008. BASE: An Acid Alternative: In Partitioned Databases, Trading Some Consistency for Availability Can Lead to Dramatic Improvements in Scalability. *Queue* 6, 3 (may 2008), 48–55. https://doi.org/10.1145/1394127.1394128

[84] Psyxto. [n. d.]. *Eventual consistency data orchestration in microservices RabbitMq broker.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/71160328/eventual-consistency-data-orchestration-in-microservices-rabbitmq-broker

[85] Maciej Pszczolinski. [n. d.]. *DB Transaction and Integrations Events dispatch - how to make it atomic?* Retrieved July 29, 2024 from https://stackoverflow.com/questions/62364508/db-transaction-and-integrations-events-dispatch-how-to-make-it-atomic

[86] raduone. [n. d.]. *Can the consumer slow down the producer, if the producer is in a different service, using Reactor Kafka?* Retrieved July 29, 2024 from https://stackoverflow.com/questions/57181251/can-the-consumer-slow-down-the-producer-if-the-producer-is-in-a-different-servi

[87] Milad Raeisi. [n. d.]. *Microservice design: change on data requirements or schema.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/68781332/microservice-design-change-on-data-requirements-or-schema

[88] Rafiq. [n. d.]. *How to handle multiple update event when there is more then one replica of a pod.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/67612615/how-to-handle-multiple-update-event-when-there-is-more-then-one-replica-of-a-pod

[89] Francisco Ramírez, Carlos Mera-Gómez, Rami Bahsoon, and Yuqun Zhang. 2021. An empirical study on microservice software development. In *2021 IEEE/ACM Joint 9th International Workshop on Software Engineering for Systems-of-Systems and 15th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (SESoS/WDES).* IEEE, IEEE, Madrid, Spain, 16–23.

[90] Tharindu Ranasingha. [n. d.]. *Is it wise to stream file over vertx event bus.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/51665264/is-it-wise-to-stream-file-over-vertx-event-bus

[91] Rasmond. [n. d.]. *What to do when exception is thrown after state is modified?* Retrieved July 29, 2024 from https://stackoverflow.com/questions/73928296/what-to-do-when-exception-is-thrown-after-state-is-modified

[92] Chris Richardson. [n. d.]. *A pattern language for microservices.* Retrieved October 1, 2023 from https://microservices.io/patterns/index.html

[93] Chris Richardson. 2022. *Pattern: Saga.* Chris Richardson Consulting, Inc. https://microservices.io/patterns/data/saga.html (Accessed on 2022-03-02).

[94] Robert. [n. d.]. *Event publisher for ASP.NET Web Api.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/39305118/event-publisher-for-asp-net-web-api

[95] sam. [n. d.]. *Refreshing microservice data with event messages.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/64049385/refreshing-microservice-data-with-event-messages

[96] Santosh. [n. d.]. *Best way to track/trace a JSON Object (a time series data) as it flows through a system of microservices on a IOT platform.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/63699141/best-way-to-track-trace-a-json-object-a-time-series-data-as-it-flows-through-a

[97] Airbnb Engineering & Data Science. 2018. *Measuring Transactional Integrity in Airbnb's Distributed Payment Ecosystem.* https://medium.com/airbnb-engineering/measuring-transactional-integrity-in-airbnbs-distributed-payment-ecosystem-a670d6926d22 (Accessed on 2023-02-25).

[98] Airbnb Engineering & Data Science. 2019. *Avoiding Double Payments in a Distributed Payments System.* https://medium.com/airbnb-engineering/avoiding-double-payments-in-a-distributed-payments-system-2981f6b070bb (Accessed on 2021-03-08).

[99] C.B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* 25, 4 (1999), 557–572. https://doi.org/10.1109/32.799955

[100] Sesigl. [n. d.]. *Use Kafka to increase resilience of a new service.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/48912603/use-kafka-to-increase-resilience-of-a-new-service

[101] Natan Silnitsky. 2022. *Event Driven Architecture — 5 Pitfalls to Avoid.* Wix. Retrieved July 1, 2023 from https://medium.com/wix-engineering/event-driven-architecture-5-pitfalls-to-avoid-b3ebf885bdb1

[102] Natan Silnitsky. 2022. *Troubleshooting Kafka for 2000 Microservices at Wix.* Wix. Retrieved July 1, 2023 from https://medium.com/wix-engineering/troubleshooting-kafka-for-2000-microservices-at-wix-986ee382fd1e

[103] Inc. Stack Exchange. 2023. *Stack Exchange Data Dump.* Stack Exchange, Inc. Retrieved September, 2023 from https://archive.org/details/stackexchange

[104] Stack Exchange Inc 2023. *Stack Overflow.* Stack Exchange Inc. Retrieved July, 2023 from http://www.stackoverflow.com/

[105] Stoffelchen. [n. d.]. *ActiveMQ 5.16.1 - Messages get stuck.* Retrieved July 29, 2024 from https://stackoverflow.com/questions/67565608/activemq-5-16-1-messages-get-stuck

[106] Ruoyu Su, Xiaozhou Li, and Davide Taibi. 2024. From Microservice to Monolith: A Multivocal Literature Review. *Electronics* 13, 8 (2024), 1452.

[107] Johnathon Sullinger. [n. d.]. *Microservices + CQRS implementation*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/62142695/microservices-cqrs-implementation

[108] Carl Thomas. [n. d.]. *How to authenticate async event messaging between services in microservice architecture*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/62109564/how-to-authenticate-async-event-messaging-between-services-in-microservice-archi

[109] TrueWill. [n. d.]. *Ensuring that all messages have been read from Kafka topic using REST Proxy*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/57222357/ensuring-that-all-messages-have-been-read-from-kafka-topic-using-rest-proxy

[110] Umbraco. [n. d.]. *What is a Staging Environment?* Retrieved June, 2024 from https://umbraco.com/knowledge-base/staging-environment/

[111] user2079172. [n. d.]. *Long running REST API with queues*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/33009721/long-running-rest-api-with-queues

[112] user3154653. [n. d.]. *Microservice Architecture dependency*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/43378165/microservice-architecture-dependency

[113] Muhammad Usman, Simone Ferlin, Anna Brunstrom, and Javid Taheri. 2022. A Survey on Observability of Distributed Edge & Container-Based Microservices. *IEEE Access* 10 (2022), 86904–86919. https://doi.org/10.1109/ACCESS.2022.3193102

[114] Guilherme Vale, Filipe Figueiredo Correia, Eduardo Martins Guerra, Thatiane de Oliveira Rosa, Jonas Fritzsch, and Justus Bogner. 2022. Designing microservice systems using patterns: an empirical study on quality trade-offs. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. IEEE, IEEE, Honolulu, HI, USA, 69–79.

[115] Nicolas Viennot, Mathias Lécuyer, Jonathan Bell, Roxana Geambasu, and Jason Nieh. 2015. Synapse: A Microservices Architecture for Heterogeneous-Database Web Applications. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 21, 16 pages. https://doi.org/10.1145/2741948.2741975

[116] VoiceOfUnreason. [n. d.]. *Aggregates in Event Sourcing Pattern*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/49985156/aggregates-in-event-sourcing-pattern/49986434#49986434

[117] Pavel Voronin. [n. d.]. *What are the strategies/frameworks for adding new services to the system where integration is based on asynchronous messages? [closed]*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/56851950/what-are-the-strategies-frameworks-for-adding-new-services-to-the-system-where-i

[118] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. 2021. Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering* 26, 4 (2021), 63.

[119] Muhammad Waseem, Peng Liang, Aakash Ahmad, Arif Ali Khan, Mojtaba Shahin, Pekka Abrahamsson, Ali Rezaei Nasab, and Tommi Mikkonen. 2023. Understanding the Issues, Their Causes and Solutions in Microservices Systems: An Empirical Study. arXiv:2302.01894 [cs.SE]

[120] Muhammad Waseem, Peng Liang, and Mojtaba Shahin. 2020. A systematic mapping study on microservices architecture in devops. *Journal of Systems and Software* 170 (2020), 110798.

[121] Jinfeng Wen, Zhenpeng Chen, Yi Liu, Yiling Lou, Yun Ma, Gang Huang, Xin Jin, and Xuanzhe Liu. 2021. An Empirical Study on Challenges of Application Development in Serverless Computing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 416–428. https://doi.org/10.1145/3468264.3468558

[122] Laurent Wilfred. [n. d.]. *Microservice archtecture data sharing/management*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/53100208/microservice-archtecture-data-sharing-management

[123] xm22. [n. d.]. *Messaging: How do your messages look like*. Retrieved July 29, 2024 from https://stackoverflow.com/questions/53025888/messaging-how-do-your-messages-look-like

[124] Xin-Li Yang, David Lo, Xin Xia, Zhi-Yuan Wan, and Jian-Ling Sun. 2016. What security questions do developers ask? a large-scale study of stack overflow posts. *Journal of Computer Science and Technology* 31 (2016), 910–924.

[125] Yang Yang, Zhifeng Chen, Qichao Chu, Haitao Zhang, and George Teo. [n. d.]. *Enabling Seamless Kafka Async Queuing with Consumer Proxy*. Uber Technologies Inc. https://www.uber.com/en-SE/blog/kafka-async-queuing-with-consumer-proxy/

[126] K. Youssefi and E. Wong. 1979. Query Processing In A Relational Database Management System. In *Fifth International Conference on Very Large Data Bases, 1979.* 409–417. https://doi.org/10.1109/VLDB.1979.718156

[127] He Zhang, Shanshan Li, Zijia Jia, Chenxing Zhong, and Cheng Zhang. 2019. Microservice architecture in reality: An industrial inquiry. In *2019 IEEE international conference on software architecture (ICSA)*. IEEE, IEEE, Hamburg, Germany, 51–60.

[128] Xin Zhou, Shanshan Li, Lingli Cao, He Zhang, Zijia Jia, Chenxing Zhong, Zhihao Shan, and Muhammad Ali Babar. 2023. Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry. *Journal of Systems and Software* 195 (2023), 111521. https://doi.org/10.1016/j.jss.2022.111521

[129] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Transactions on Software Engineering* 47, 2 (2021), 243–260. https://doi.org/10.1109/TSE.2018.2887384

[130] Olaf Zimmermann. 2017. Microservices Tenets. *Comput. Sci.* 32, 3-4 (2017), 301–310.