Data Management in Microservices: State of the Practice, Challenges, and Research Directions

Nunes Laigner, Rodrigo; Zhou, Yongluan; Vaz Salles, Marcos Antonio; Liu, Yijian; Kalinowski, Marcos

Data Management in Microservices: State of the Practice, Challenges, and Research Directions (Extended Version)

Rodrigo Laigner
University of Copenhagen
Copenhagen, Denmark
rnl@di.ku.dk

Yongluan Zhou
University of Copenhagen
Copenhagen, Denmark
zhou@di.ku.dk

Marcos Antonio Vaz Salles
University of Copenhagen
Copenhagen, Denmark
vmarcos@di.ku.dk

Yijian Liu
University of Copenhagen
Copenhagen, Denmark
liu@di.ku.dk

Marcos Kalinowski
PUC-Rio
Rio de Janeiro, Brazil
kalinowski@inf.puc-rio.br

ABSTRACT

We are recently witnessing an increased adoption of microservice architectures by the industry for achieving scalability by functional decomposition, fault-tolerance by deployment of small and independent services, and polyglot persistence by the adoption of different database technologies specific to the needs of each service. Despite the accelerating industrial adoption and the extensive research on microservices, there is a lack of thorough investigation on the state of the practice and the major challenges faced by practitioners with regard to data management.

To bridge this gap, this paper presents a detailed investigation of data management in microservices. Our exploratory study is based on the following methodology: (i) we conducted a systematic literature review of articles reporting the adoption of microservices in industry settings, where more than 300 articles were filtered down to 11 representative studies; (ii) we analyzed a set of 9 popular open-source microservice-based applications, selected out of more than 20 open-source projects; furthermore, (iii) to strengthen our evidence, we conducted an online survey that we then used to cross-validate the findings of the previous steps with the perceptions and experiences of over 120 practitioners and researchers.

Through this process, we were able to categorize the state of practice and reveal several principled challenges that cannot be solved by software engineering practices, but rather need system-level support to alleviate the burden of practitioners. Based on the observations we also identified a series of research directions to achieve this goal. Fundamentally, novel database systems and data management tools that support isolation for microservices, which include fault isolation, performance isolation, data ownership, and independent schema evolution across microservices must be built to address the needs of this growing architectural style.

1 INTRODUCTION

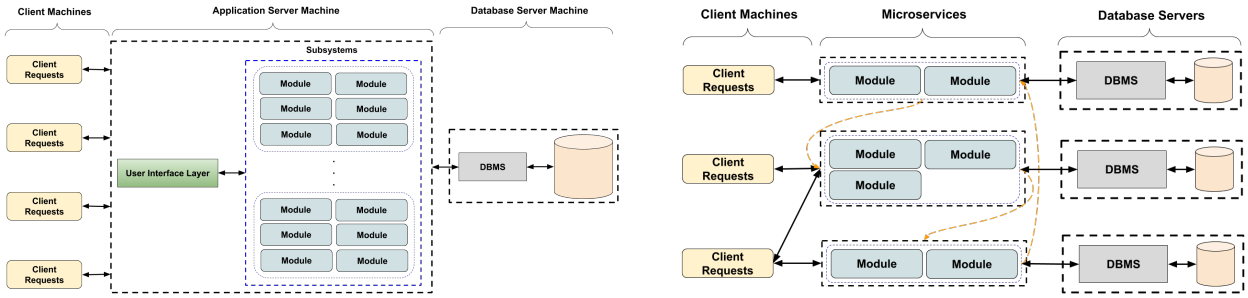
The advent of large-scale online services provoked an architectural shift in the design of data-driven applications, with resulting needs for designing distributed application systems from the point of views of both computational resources and software development team organization [21, 57]. In particular, we are witnessing the increasing adoption of microservice architectures to replace the traditional monolithic architecture (Figure 1(a)).

In contrast to a monolithic architecture, where modules and/or subsystems are integrated and cooperate in a centralized manner, a microservice architecture, as depicted in Figure 1(b), organizes an application as a set of small services that are built, deployed, and scaled independently. Microservices communicate with each other through lightweight mechanisms, such as HTTP-based protocols [21, 25] or asynchronous messages [83]. In a microservice-based application, a new functionality or bug fix does not require a new build and subsequent deployment of the whole application, but instead, it can be managed by redeployment of a microservice unit. In the same line of reasoning, microservices also enable design for failure. As faults in individual microservices are isolated, their propagation to other building blocks of the architecture is limited [83].

Furthermore, each microservice may also manage its own database that is suitable to the data formats and workloads of the microservice. This flexibility is often associated with the polyglot persistence principle, where different categories of database systems (e.g., loosely structured NoSQL vs. relational) service separate microservices [20]. Thus, a microservice architecture represents a significant shift from traditional monolithic transaction processing systems. In particular, in the monolithic architectural style, transactions can be easily executed across modules, while, in microservices, it becomes necessary to break these transactions down due to the decomposition of the application in small parts.

Motivation. Despite the increased adoption of microservices in industry settings [3, 16, 29, 30, 41, 46, 49, 50, 66] and the perception that data management is a major challenge in microservices [23, 40, 47, 79, 82], there is little research on the characteristics of data management in microservices in practice. Besides, existing studies provide a limited investigation of the major challenges practitioners face regarding data management in such an architectural style. For instance, it is unclear which database technologies and patterns are adopted, which data consistency semantics are employed within and across services, or which mechanisms are in fact used to exchange data in these architectures. Understanding these issues would provide valuable insights as to how to advance data management technologies to meet the needs of microservice applications.

Contributions. To bridge this gap, this paper presents an investigation of the state of the practice of data management in microservices.



(a) Traditional monolithic architecture

(b) Microservice architecture

Figure 1: Monolithic vs. microservice architectures

Specifically, we perform an exploratory study based on the following methodology: (i) we systematically review the literature on articles reporting the adoption of microservice architectures; (ii) we further analyze popular open-source repositories of microservice-based applications; and (iii) we design an online survey in order to gather the opinion of developers and researchers experienced with microservices in real world settings, allowing us to cross-validate with the findings of the previous steps. Taken together, these three interrelated explorations provide new and comprehensive evidence on data management practices and challenges in microservice architectures.

From our investigation, we revealed that microservice developers are dealing with a plethora of data management challenges. While microservices are supposed to work autonomously, they often surprisingly end up exhibiting functional dependencies amongst each other and depending on each others' private states. Practitioners are performing business transaction processing and resorting to the formation of data processing pipelines in an ad-hoc way by chaining microservices together. As a result, developers have a hard time reasoning about enforcement of application safety within and across microservices, which relate to challenges in managing constraints over distributed microservice states, reasoning about the unintended interleaving of event streams, dealing with weak concurrency isolation, enforcing data replication semantics, and ensuring consistency across a variety of storage technologies.

Practitioners are poorly served by state-of-the-art database systems and end up employing ad-hoc and suboptimal solutions to meet data management requirements in such architecture. As a result of substantial encoding of data management logic in the application-tier, database systems no longer play a central role in this novel paradigm. The lack of a holistic view, by means of an unawareness of the dataflow, constraints, and the complex interplay of microservices, leads to the impossibility of effectively ensuring application safety in microservice paradigm. Based on the revealed challenges, we identified a set of research directions for future database systems so they can play a central role in this new paradigm.

Related Work. Despite the extensive work in microservice architectures, existing studies lack an in-depth analysis of data management, rather focusing on migrating from monolithic architecture to microservices [9, 10] or investigating other general software engineering aspects [23, 27, 40, 79, 82], such as software attributes (e.g., coupling and cohesion). To the best of our knowledge, our work is a first step towards understanding how microservice developers interact with the database systems that this community builds, fostering further research into these issues.

In addition, our work provides an in-depth characterization of challenges faced by microservice developers while encoding data management logic in the application-tier. In regard to this contribution, although some studies described related pitfalls, such as *shared persistence* [61, 74], and previous literature investigated architectural smells and anti-patterns in microservices [9, 56, 75], they fail to capture properties of consistency models and technical issues of database systems, such as data replication and constraint enforcement, as we provide in this work.

Most important, in light of gist data management challenges in microservice architectures, our work is the first to reflect on core database limitations that fail to serve the needs of this emerging architectural style.

Outline. This paper is structured as follows. Section 2 details the methodology. Section 3 discusses the state of practice regarding data management in microservices. Sections 4 and 5 discusses challenges faced by practitioners while developing data-intensive microservices. The challenges result in research directions that are explored in Section 6. Finally, Section 7 addresses our conclusion.

2 METHODOLOGY

In this section, we explain the methodology defined to allow an in-depth analysis of the state of practice of data management in microservice architectures.

Literature Review. We opted to start with the collection of information from the literature, following the guidelines of [38]. A total of more than 300 papers retrieved from different digital libraries, such as Scopus and Scholar, were analyzed and a set of 10 papers [3, 14, 16, 30, 41, 42, 49, 50, 66, 78] were able to meet our selection criteria. Even though very few papers provided detailed enough information about data management in microservices, we believe these studies are strongly representative of the peer-reviewed literature.

Analysis of Open-source Repositories. For our analysis, our goal is to include a heterogeneous body of strategies for data management in microservices (e.g., including different databases, domains, and technologies). In regard to the quality criteria, we aimed at projects that exhibit historical developer engagement and pose characteristics that allow us to characterize data management challenges (e.g., presence of data management logic). We relied on

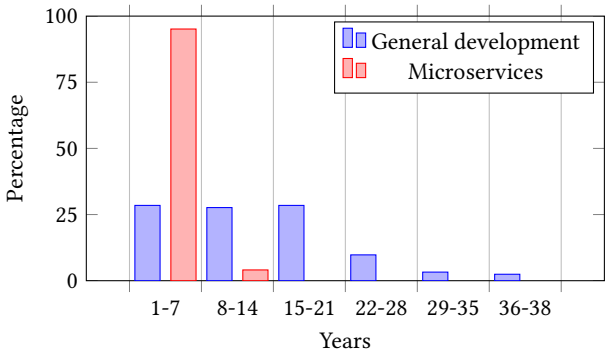


Figure 2: The participants' experience

Table 1: Sizes of the participants' organizations

Size	1-10	11-100	101-1000	1001-10000	> 10000
%	10.57	19.51	31.71	22.76	15.45

exploratory searches on GitHub and catalogued data sets of open-source repositories [34, 54, 55, 73]. In total, 9 open-source projects [2, 18, 37, 52, 53, 69, 70, 72, 81] were selected for further analysis.

Expert Survey. To further detail the state of the practice of microservices concerning data management as well as to identify emerging trends among practitioners and researchers, we designed an online survey. We considered multiple methods to recruit participants, such as direct emails, LinkedIn, Twitter, Facebook and Google groups, Slack Channels, and discussion lists (e.g., meetup groups). The questionnaire was designed with 27 questions grouped into 3 high-level categories: (i) background information about the participants; (ii) state of the practice of data management in microservices (e.g., databases, computations, deployments, consistency semantics, and application invariant validations); and (iii) challenges faced. We give an overview of the experience, the sizes of organizations, and the roles of the participants as follows.

Figure 2 shows the experience of the participants in general software development (not only microservices) and experience with microservice architectures. The results demonstrate the large general experience of the respondents, which contrasts with the experience in microservices. The latter is concentrated in no more than 7 years among most respondents, indicating a new industry trend. Nevertheless, 83.87% of the participants have been involved in at least one microservice project in industry.

Table 1 exhibits the sizes of the organizations of the participants. Although some may assume microservices are prevalent exclusively in large Internet-scale firms, we highlight that microservices are also adopted in small and medium-sized organizations.

We asked the participants about their current role at work. This question is important to identify the positions of microservice adopters in firms. We provided them with a set of choices, along with an *Other* option, so they could indicate an additional role. The top five roles were back-end developer (29.27%), software architect (17.89%), researcher (17.89%), full-stack developer (15.45%), and manager (7.32%). Amongst other roles selected, we highlight DevOps, CTO, and Project Manager.

Lastly, we collected from the participants the application domains of the microservice applications that they have developed or maintained over time. The respondents indicated more than 15 different domains, including Retail & E-commerce (33%), Finance (21%), Education (16%), Banking (14%), Telecommunications (11%), Smart Cities (10%), Transportation, Marketing, GIS, and more. Similar patterns are also observed in papers from literature and open-source repositories. *The results suggest that technical solutions to data management in microservice architectures should be designed to be applicable across domains.*

Table 2: Motivations for microservices in data management

Motivation	#
Scalability through functional decomposition	55
Fault-isolation (e.g., increasing data availability)	32
Agility on data change (e.g., facilitating schema evolution)	32
To enable event-driven data management (e.g., as opposed to classic pull-based data querying)	23
Polyglot persistence	9
Others	3

3 STATE OF THE PRACTICE

In this section, we focus on characterizing how microservice-based applications are being designed and deployed in practice with a particular focus on data management.

3.1 Motivations

Existing works [10, 21, 25, 30, 57] address that the major reasons for adopting microservices are related to fault-isolation, independent software evolution (including schema evolution), and scalability of individual system components. By investigating several microservice deployments, we were able to reveal that such desirable properties are enabled by data partitioning and decentralized data management (by means of database/schema per microservice). Thus, these motivations are intrinsically related to data management, and decentralized data management is a major foundation in microservices adoption.

To investigate the most compelling directions for future avenues of research in data management for microservices, we asked the survey participants to select the top 2 reasons to adopt a microservices architecture regarding data management. The 5 options given were centered on data management concerns (i.e., no software engineering concerns, such as loose coupling and easier maintenance, were considered) and were influenced by discussions among the authors, the literature review, and the analysis of open-source repositories. Table 2 shows the options provided and respective responses. We highlight the following important observations.

Functional partitioning: To support scalability (i.e., spreading functional groups across databases) and high data availability (i.e., achieving functional isolation of errors), functional decomposition of the application is a major driver for adopting microservices according to both survey respondents (57%) and the literature [14, 16, 30, 42, 49, 50, 66]. *These results suggest that the design of data management technologies for microservices should focus not only on scalability, but also on stronger mechanisms for fault-tolerance and error isolation.*

Though not explicitly mentioned by literature, we deduce from our analysis that functional decomposition is reminiscent of the idea of functional scaling introduced by Pritchett [63], a strategy

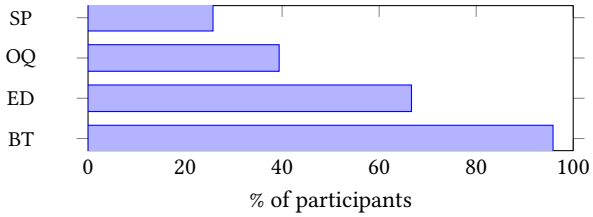


Figure 3: Computations indicated by participants

that involves "grouping data by function and spreading functional groups across [different] databases."

Decentralized data management: Practitioners developing applications in real-world settings largely deal with evolving requirements and subsequent schema changes [78]. The ability of microservice architectures to provide independently-evolving schemas in different services, in contrast with the unified schema of monolithic architectures, is another major driver [14, 30, 41, 49, 78].

Surprisingly, although polyglot persistence derives naturally from decentralized data management, it is the least cited motivation in both literature [3, 41, 78] and the survey. *These results suggest that better tools for schema evolution in microservices are a more pressing need than the support for a large variety of data models.*

Event-driven microservices: Event-driven (a.k.a. reactive) systems constitute an emerging trend in the design of data-driven software applications [7]. Microservices are often mentioned as a compelling paradigm for designing reactive applications [36, 68, 71]. A set of 23 participants consider event-driven data management a primary motivation for microservices, which may indicate a trend in industry adoption. Most papers [14, 30, 41, 42, 49, 50, 66, 78] in the literature mention the use of asynchronous primitives for message-oriented communication to achieve loose coupling among microservices and decentralized data management. However, while open-source repositories [2, 37, 53, 69, 70] present use cases that rely on the generation of event streams for further processing, using microservices for stream processing is not significantly covered in literature. *This observation suggests that data streaming should emerge as a concern for the literature in microservices.*

3.2 Computations

We were also interested in characterizing the types of queries and data computations that are performed in microservice-based applications. Towards this aim, a preliminary assessment of the survey was conducted with three developers with experience in microservice architectures. We realized that they were not familiarized with database terms used in academic research, such as analytical queries and online transaction processing (OLTP).

Thus, to avoid the threat of misconceptions in answers, we defined a set of open and multiple-choice questions to obtain from the participants relevant information about the computations performed in their applications. Figure 3 shows the types of computations identified from the responses, characterized by Cross-Microservice Business Transactions (BT), Event-Driven Computing (ED), Online Queries (OQ), and Stream Processing (SP).

3.2.1 *Business transactions across microservices.* [1]

[1]Our use of the term "business transaction" in this paper refers to any unit of work carried out by an application [59], not necessarily under ACID semantics.

The studies reviewed from literature [16, 30, 50, 66, 78], the open-source repositories [2, 18, 37, 52, 53, 69, 72, 81], and the use cases described by participants indicate the prevalence of decentralized OLTP-like workloads in microservice-based applications, such as tracking orders in web shop applications.

Evidence from literature and open-source repositories. By analyzing the open-source repositories, we observed that OLTP-like intra-microservice transactions are common, such as those found in conventional applications [4, 13]. Similar to Pavlo's findings [60], we did not find evidence of the use of stored procedures or the configuration of databases in serializable isolation. The same applies to literature. *The decentralized data management principle makes shipping stored-procedures and assuming data is held in a single database unfit for this architectural paradigm.*

Regarding computations performed across microservices, conversations (i.e., the interaction between a set of consumer and provider services) are prevalent. Hohpe [33] argues that orchestration and choreography are the two types of conversations that take place in the context of distributed web applications. Most papers [3, 30, 41, 42, 49, 50, 66] and open-source projects [2, 18, 37, 52, 69, 70, 72, 81] report the use of the choreography conversation pattern [33] through both synchronous workflows and asynchronous event-based workflows.

In open-source repositories, the latter is dominant, implying that updates affecting other microservices are queued for asynchronous processing. This finding has led us to observe that microservice architectures are indeed reminiscent of BASE [63], which targets functionally decomposing an application to achieve higher scalability in exchange for a weak consistency model. The same trend is found in the literature. Besides, some papers report the use of orchestration [3, 14, 16]. Only one open-source project [53] adopts a saga-like orchestration. All of the options above are characterized by weak consistency models [6] where on an operation that span multiple microservices, the tasks may complete at any point in time, and the data returned is only eventually consistent.

While the literature [21, 57, 83] mentions the principle that microservices are autonomous components that are independently deployed and evolved, we observed that most microservice-based applications often perform operations that span multiple microservices [19], which indicates a functionality dependence between microservices.

In sync with recent discussions in the community [31, 60], we did not find any evidence of distributed transactions, such as through the 2-Phase Commit (2PC) protocol, in both the open-source repositories and the reviewed papers from literature.

Evidence from industry settings. To further characterize the implementation of business transactions in microservices, we focused on understanding how consistency guarantees are enforced in industrial settings. We asked the participants which mechanisms to coordinate operations spanning multiple microservices they have been employing. The options were defined based on the patterns found in the literature and open-source repositories, along with a preliminary assessment of the survey. Table 3 shows the responses sorted in descending order.

[2]Some respondents were selecting the option *distributed transactions* but their subsequent answers were not compatible with distributed commit protocols (e.g., employing events to trigger actions asynchronously as state becomes consistent in a microservice).

Table 3: Mechanisms for inter-microservice coordination

Coordination mechanism	#
Orchestration	37
Sagas (centralized approach, with a Saga coordinator)	24
The Back-end for Front-end Pattern (BFF)	24
Choreography	22
Sagas (decentralized approach, i.e., no Saga coordinator)	14
Distributed transactions (e.g., via 2PC) [2]	14
2-Phase Commit	11
Others	16

In contrast with findings from literature and open-source repositories, orchestration-like (including sagas [26] and the backed-end for front-end pattern (BFF)[3] [8]) mechanisms are the most popular in industry settings. To further understand these orchestration-like mechanisms, we asked the participants which orchestration engines they used to support operations spanning multiple microservices. The results highlight that the adoption of custom-made (e.g., company-built) orchestration engines is prevalent among participants (51.2%).

We also asked the participants to briefly describe one of their use cases involving consistency in operations spanning multiple microservices. Most responses indicated the implementation of workflows through application code, and the use of application-level validations to safeguard the constraints of the workflow.

Despite the prevalence of orchestration-like mechanisms, choreographies are also highly mentioned. In this context, an interesting quote provided by one of the respondents characterizes this practice in industry, which is well-aligned with previous findings:

> I am absolutely against the business logic inside the database. Depending on the scale I would refrain from using transactions at all, favoring an event-driven approach, with eventual consistency and micro-transactions.

The main difference between the orchestration-like mechanisms described by the participants and the choreography mechanisms found in the open-source repositories, literature, and participants is the type of communication. The former is through HTTP-based and synchronous requests, whereas the latter is mostly event-based and asynchronous.

The responses related to choreography are consistent with the literature and open-source repositories, since most papers [14, 16, 30, 49, 50, 66, 78] position eventual consistency as the de facto consistency model in microservice applications and the adoption of BASE-like functionally decoupled transactional operations [2, 18, 37, 52, 53, 69, 70, 81].

The prominence of orchestration engines in industry settings may indicate the preference for solutions that allow for finer-grained control, debugging, and observability over the operations spanning multiple microservices.

We then opted to change the option to *2-Phase Commit*. By analyzing such answers, we estimate that 11 of the responses provided are not compatible with distributed transactions. This leads choreography to 33 responses in total.

[3]In this mechanism, a service centralizes the role of performing requests across microservices.

3.2.2 Online queries. While one may argue that queries spanning multiple microservices are antagonistic to the principles of state encapsulation and independent data silos of microservices, we found abundant evidence of such cases in the literature [3, 14, 42, 78] and open-source repositories [2, 53, 69, 70, 72]. Therefore, to further understand this trend, we asked the survey participants to identify if they have implemented queries aggregating data from multiple microservices and to describe one of their use cases. 26 participants declared the use of some mechanism and 21 provided a short-answer describing it. We unveiled three mechanisms to allow for such queries and explain them as follows.

A. Queries aggregating data belonging to different microservices. In this mechanism, a consumer service contacts, often through an HTTP request, a set of microservices through their APIs. After receiving all responses, the consumer service then aggregates the data in-memory (also performing joins, if necessary) and serves the client. We identified the following three practices to implement such a mechanism: (i) Through composition of service calls, i.e., a microservice performs the necessary synchronous requests to retrieve data from other microservices. Six respondents described the adoption of such practice; (ii) One respondent declared the use of the BFF pattern [8]. We also identified such practice in a repository [2]; (iii) Lastly, one respondent declared the use of the API Gateway pattern [65]. We also observed its adoption in open-source repositories [53, 69, 72] and the literature [3, 41].

From the respondents' answers, we could not observe significant differences between the BFF and the API Gateway patterns in terms of query serving.

B. Replication. We also unveiled the use of ad-hoc mechanisms for data replication across microservices for online querying purpose. We explain the identified practices as follows.

(a) *Replication across microservices.* This practice is characterized by a microservice generating events related to its own updated data items and communicating these changes asynchronously, often through persistent messaging supported by a message broker.

We identified the prevalence of weak delivery semantics in open-source repositories, i.e., although updates to the same object are often sequentially ordered by the publisher, there is no ordering guarantee regarding updates to different replicated objects. In other words, causal dependencies are ignored on updating replicated data items. The responses provided by participants suggest the same. This choice appears to be consistent with the eventual consistency semantics adopted by the synchronous query mechanisms described above.

(b) *Replication to a database.* This practice is characterized by two mechanisms: (i) Daemon workers, one for each microservice and its respective generated events, or a central service, are responsible for subscribing to data item updates and replicating these to a special-purpose database used for querying; (ii) this practice is also characterized by the use of batch workers (usually special-purpose microservices) to extract data from microservices periodically (with a pull-oriented strategy) and replicate those in a neutral data repository for fast querying (e.g., ElasticSearch). We suspect the second approach is reminiscent of the behavior of Extraction-Transform-Load (ETL) tools [77]. Although it is unknown why ETL tools are not being employed for such task, we believe the dynamicity provided by the autonomous deployment of microservices plays a role.

(c) Lastly, the use of data stream processing systems (DSPSs) for processing streams (e.g., updates to data items) generated by microservices to build materialized views was also mentioned. One respondent declared: "[...] materializing views over various time windows, making them queryable to other services."

C. Views. While service composition and replication are often subject to the adoption of the database per microservice pattern (Section 3.3), when microservices share the same database, practitioners may rely on views across multiple schemas to serve cross-microservice queries. This is the least cited practice.

3.2.3 Stream processing. To understand how data stream processing systems (DSPSs) that the database community builds interact with microservice architectures, we asked the participants if they have already employed a DSPS in conjunction with microservices. We also asked them to provide a description of one of their use cases in a short answer. 17 out of 49 respondents declared the use of DSPSs and 12 provided a short answer.[4] We summarize these as follows.

Data processing pipelines. We observed the use of application libraries targeted at stream processing [35] in microservices to perform data transformations, as mentioned by one participant: "We use Kafka Streams to reorder (time window) out-of-sequence data[.]" Besides, one respondent declared the formation of an "ETL chain implemented with microservices that exchange information asynchronously using Kafka as a message bus. [...] We have several microservices in an ETL chain, [... including] several transformation steps." The person explains that "the chain will fork at a certain point and one side of the fork will carry one the transformation up to message delivery to downstream systems, the other side of the fork will do asynchronous writes to an operational data store (a NoSQL database). [..] These writes are asynchronous to remove the DB from the critical path and ensure that messages can be delivered in near-realtime."

These responses represent a surprising trend, since we did not find substantive evidence of the use of microservices for forming a data processing pipeline in the literature and open-source repositories. Existing literature suggests an impedance between microservices and DSPSs [1, 36, 80] particularly related to the stream processing abstraction. Often in the form of static dataflow graphs, operations such as filter, join, and aggregate are applied uniformly to all stream items in such abstraction. This model notably contrasts with microservice principles, including loose-coupling, fault-isolation, independent evolution, and autonomous deployment.

Most important, the dynamic behavior of microservices, including operating over data items from different microservices, introduce a significant challenge to express complex business logic using this abstraction. *Thus, the responses suggest microservice developers find the static dataflow abstraction difficult to express their computations. As a result, they end up relying on the ad-hoc formation of data processing pipelines by microservices.*

Anomaly detection. The use of DSPSs for anomaly detection based on event streams generated by microservices was mentioned by two respondents. For instance, one respondent declared the adoption of a stream processing engine for "monitoring financial operations by analyzing situations and generating critical events for microservices that convert these events into alerts."

Replication and materialized views. As already mentioned in the last section, four respondents indicated the use of stream processing engines to process data generated from microservices aiming at replicating data and materializing views.

3.2.4 Event-driven computing. As mentioned previously, we observed that microservice architectures are often reminiscent of BASE [63]. Pritchett [63] argues that organizing computation in an event-driven architecture (EDA) may allow further scalability and architectural decoupling improvements. In an EDA, events that are relevant to an incoming request are generated when a consistent state is reached to allow further processing [63].

To characterize how EDA intersects with microservice architectures, we asked the participants to declare whether they have performed event-driven computations in microservices through an EDA and to provide an example of one of their use cases in a short answer. 45 participants indicated the use of such computations and 26 provided a brief description of an use case.

Overall, the responses are consistent with the findings from the literature and the open-source repositories. EDA is often an enabler of event-based and asynchronous workflows. Some quote snippets are provided as follows: (a) "Pure choreography without central orchestrator;" (b) "natural way for microservices to collaborate when they depend on data from other microservices;" (c) "Payments system using an [EDA] to process ecommerce orders/payments." (d) "[...] to trigger several microservices in our architecture."

3.3 Database and Deployment Patterns

There are three mainstream approaches for using database systems in microservice architectures: (i) private tables per microservice, sharing a database server and schema; (ii) schema per microservice, sharing a common database server; and (iii) database server per microservice [51]. We asked the participants which database patterns they have been using to support data management in their microservices. We also asked the participants what drivers led to the adoption of each database pattern in their microservice architectures.

The results show that most practitioners prefer encapsulating a microservice state within its own managed database server (43% of responses). The same trend is observed in the literature and open-source repositories. The participants indicated the following drivers that lead to this adoption: (i) Achieving loosely-coupled microservices; (ii) Independent data layer scalability. The ability to scale each microservice's underlying database independently, which would otherwise be challenging with a single database server supporting multiple (heterogeneous) tenant applications; and (iii) fault-isolation, which is naturally derived from the already mentioned formation of independent silos of data.

Besides, practitioners, literature, and open-source analysis indicate that container-based deployment is the de-facto practice. Each microservice and respective database are bundled in separate containers, thus guaranteeing that each can be scaled independently and faults are limited to the container boundary. *The results suggest data management system must embrace these patterns to better serve data-intensive microservices.*

[4] It is worthy to note that only one paper mentioned the use of a stream processing engine [42].

3.4 Database Systems

We asked the participants what database systems they have been adopting in their microservice-based applications. Our objective was to understand the types of database technologies adopted, specially concerning the data model, performance and scalability aspects. The results are shown in Figure 4. It is worthy to note that we merged the MySQL and MariaDB responses.

It is often reported the adoption of multiple DBMS belonging to at least two of the provided categories. For instance, 47.97% indicated the use of at least one relational DBMS in conjunction with MongoDB. Besides, we observed a trend (15%) of the following stack: a relational DBMS (e.g., PostgreSQL, MySQL, or SQL Server) + Redis + MongoDB + ElasticSearch. The most common use case for this stack is the use of relational or document-oriented databases as the underlying DBMSs of microservices, the use of Redis as a cache layer to provide fast data access to recurring requests, and the replication of data through an event-driven approach to ElasticSearch for fast online analytical queries. Overall, the results of DBMSs adoption from the survey are very aligned with the reviewed papers and the open-source repositories.

4 MICROSERVICES THROUGH THE LOOKING GLASS

In the previous section, we explored how data-intensive microservices are being developed and deployed in the wild. Microservice applications deviate significantly from the architecture of traditional database applications, introducing significant data management logic at the application-tier and a decentralized model for data management.

Such findings urged us to investigate challenges faced by developers that would drive research avenues in data management. Particularly, we aim to answer whether the observed shift promotes challenges that cannot be solved by traditional database systems.

Therefore, we present next a series of challenges (referenced by **C#**) faced by developers that reveal a myriad of unmet needs that should be appropriately addressed by the database community.

4.1 Cross-microservice validations

Background. Given the ubiquity of business transactions across microservices, we start by reviewing a type of application-level validation employed by developers to ensure correctness in such cases. Take for example the snippet adapted from the project *vertx* [69] shown in Figure 5. Prior to proceeding with the client's checkout request, the *cart* microservice verifies, through a HTTP remote call to the *inventory* microservice whether the items in the cart are available. If so, it generates an event requesting the corresponding order to be processed.

However, such validation is not safe under concurrency. By the time the *Order* microservice processes the event, one (or more) of the items in the cart might not be available anymore. Under high contention such criteria may lead to abusive generation of events, resource trashing, and may also introduce the burden to deal with compensation logic in the application. It is worthy to note we encountered such pattern in several other projects, such as [2, 37, 52, 53, 72].

Discussion. In the case presented in Figure 5, coordination is a condition necessary to ensure correctness under conflicting reads and writes. However, in the absence of efficient solutions and intuitive interfaces for encoding concurrency control semantics in the application-tier [32], developers end up encountering challenges to safeguard constraints across microservices.

Another impediment that makes the problem even more difficult comes from the heterogeneous database systems encountered in microservice architectures, the incompatible isolation levels, and the corresponding lack of interoperability among them.

C1. In the absence of efficient or viable solutions for isolation guarantees in the application-tier, microservice developers are exposed to concurrency anomalies. This creates a great barrier for expressing correctness criteria across different microservices.

4.2 Implicit cross-microservice associations

Background. The functional partitioning through BASE-like approach often does not refrain developers from modeling implicit associations across microservices [5]. We encountered several cases where enforcement of foreign key constraint across microservices is a necessary condition for ensuring application correctness, and the applications analyzed show no evidence of such enforcement.

Consider the source code snippet exhibited in Figure 6 adapted from *event-stream* [37], *vertx* [69], and *eShopContainers* [2] applications. In the event of a product removal from the *product* microservice, in the absence of cascading delete operation, operations carried out by the system and records stored in other microservices that rely on the existence of the deleted product(s) will face no impact, which may bring the system to an inconsistent state.

This pattern is observed across several other microservice applications [2, 37, 52, 53, 69].

Discussion. Microservice developers have to explicitly enforce implicit foreign key constraints across microservices to refrain bringing the system into an inconsistent state, a complicate and error-prone task. However, in most cases, practitioners simply either ignore or are unaware of the consequences. Giving up foreign key constraint enforcement across microservices lead to "orphaned" records in one or more microservices. Such pitfall is even worse than encoding associations under non-serializable isolation, since it may lead to a much higher number of anomalies [4].

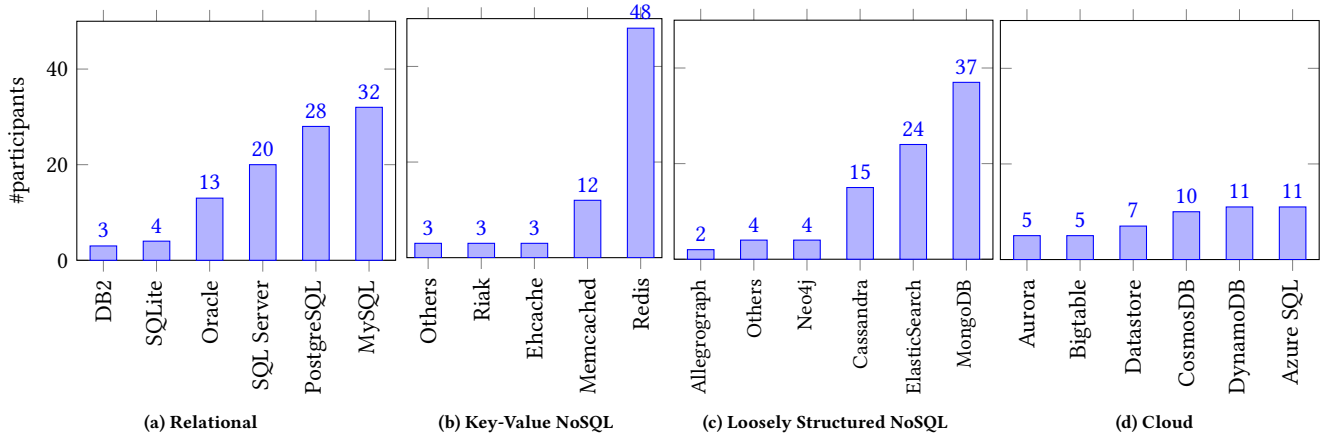
C2. The impossibility of declaring foreign key constraints between different microservices' schemas creates a great barrier for developers to ensure constraints across microservices.

4.3 Cross-microservice queries

Background. As part of our study, we observed the popularity of online queries in microservice architectures (Section 3.2). Given the distributed nature of data stores, the data encapsulation may introduce challenges not found in traditional monolithic systems.

The literature [3, 41] and open-source repositories provide interesting examples of microservices being employed for data aggregation through queries spanning different microservices (and their

[5] We refer to implicit associations those relationships between tables from different microservices that would exist if the application schema was designed as a single database.



(a) Relational **(b) Key-Value NoSQL** **(c) Loosely Structured NoSQL** **(d) Cloud**

Figure 4: Database systems adopted in microservice architectures

Figure 5: Cross-microservice validation example

```
// Cart microservice: Checking availability of products
private boolean checkAvailableInventory(ShoppingCart cart) {
    List allInventories = getInventoryEndpoint()
           .HttpGet(cart.getProductItems());
    return allInventories.map(inventories -> {
         List insufficient = inventories.filter(item ->
              item.get("inventory") - item.get("amount") <
              0).toList();
         if (insufficient.size() > 0) return false;
         else return true; });
}
```

Figure 6: Implicit cross-microservice association example

```
// Product microservice: Delete product, no cascade delete
public ProductService deleteProduct(String productId,
     Handler<AsyncResult<Void>> resultHandler) {
    this.removeOne(productId, DELETE_STATEMENT,
         resultHandler);
    return this;
}
-------------------------------------------------------------
// Cart microservice: Product items without association
public class ShoppingCart {
  private List<ProductTuple> productItems;
  private Map<String, Integer> amountMap;
  public ShoppingCart() {}
} // additional code omitted
```

underlying databases), often with different data models. Such implementations reveal a new trend on stateful middle-tier applications that are particular to microservices: encoding of data processing functionality at the application-level.

Consider the example adapted from the project FTGO [53], shown in Figure 7. The code snippet exhibits a method (*getOrderDetails*) responsible for reaching out to several microservices in order to consolidate in real-time a client view (i.e., the order details).

Discussion. Current state of the practice leaves the developer responsible for retrieving the appropriate data from each microservice and dealing with possible inconsistencies, such as fractured

Figure 7: Cross-microservice query example

```
public OrderDetails getOrderDetails(Long orderId) {
    OrderInfo orderInfo =
         orderService.findOrderById(orderId);
    TicketInfo ticketInfo = kitchenService
          .findTicketById(orderId);
    DeliveryInfo> deliveryInfo = deliveryService
          .findDeliveryByOrderId(orderId);
    BillInfo billInfo = accountingService
          .findBillByOrderId(orderId);
    Mono<OrderInfo,TicketInfo,DeliveryInfo,BillInfo>
         combined = Mono.zip(orderInfo, ticketInfo,
         deliveryInfo, billInfo);
    OrderDetails orderDetails = combined.map(
         OrderDetails::makeOrderDetails);
    return orderDetails;
}
```

reads [5], an impedance that may lead to a complex code base and bugs. Besides, with such application-level data management there is no way to ensure that reads to different microservices reflect a view of the entire application at a single point in time. In other words, transactional consistency [62] is not possible. Lastly, with such sequential request pattern, as observed in Figure 7, some (or all) requests to microservices may fail, thus leading to missing records and an incomplete result.

C3. Developers have no support on querying multiple microservice database states consistently and they end up encountering challenges that should have been solved by database systems.

4.4 Non-transactional queuing

Background. In an intra-microservice transaction, updates affecting other microservices are queued for asynchronous processing. As advocated by Pritchett [63], this queuing must be part of the transactional context of the database operation in the originating microservice. However, as unveiled in our study [44], there is no evidence of use of message persistence queues in the same resource as the database. Practitioners rely on external message persistence

Figure 8: Non-transactional queuing example

```
public ResponseEntity createInsuranceQuote() {
   InsuranceQuote insuranceQuote = new InsuranceQuote
       (date, QUOTE_SUBMITTED, insuranceOptionsEntity);
   insuranceQuoteRepository.save(insuranceQuote);
   policyMessageProducer.send(date, insuranceQuote);
   return ResponseEntity.ok(insuranceQuote);
} // parameters omitted
-------------------------------------------------------------
public void send(Date date, InsuranceQuote insuranceQuote) {
    InsuranceQuoteEvent insuranceQuoteEvent =
       new InsuranceQuoteEvent(date, insuranceQuote);
   try { jmsTemplate.send(insuranceQuoteEventQueue,
        insuranceQuoteEvent);
   } catch(JmsException exception) {
      logger.error("Failed to send", exception);
   }
}
```

queues (e.g., message brokers) without employing a distributed commitment protocol for message queuing.

Consider the example adapted from the project Lakeside Mutual [52], shown in Figure 8. The method *createInsuranceQuote* performs an insert operation in the database regarding the data item *insuranceQuote*. Afterward, a message representing the operation is queued without a transactional guarantee. In case of error, logging is performed but no additional measures are taken by the application, which continues its execution. This pattern is identified in several applications [18, 18, 37, 52, 70].

Discussion. Although existing database systems provide a backing messaging or notification persistence on the same resource as the database [17], the clients are forced to access the same database instance. As a result, most open-source projects and the literature rely on external message brokers for the communication and event exchange between microservices. Without interoperability of databases and message brokers, 2PC is not possible, violating the atomicity property prescribed by Pritchett [63].

C4. Developers lack viable and efficient abstractions for transactional queuing in microservice architectures. As a result, anomalies arise due to lack of isolation and ad-hoc fault-handling, leading to challenges on reasoning about application state.

4.5 Feral ordering

Description. Consider an application in the e-commerce domain. After adding several items to a cart, which is managed by the *cart* microservice, the customer may initiate the order's payment process through the API of the *payment* microservice. Suppose that prior to submitting the order, the *cart* microservice emits an event representing the change of the price of one of the items in the customer's cart.

Two options may apply in this case: (i) Application constraints require that events related to changes in the price of items should be reflected in the orders submitted for processing; or (ii) application constraints are such that price changes should not affect such orders. However, we noticed cases where the application does not enforce any constraint. Both options may apply depending on the eventual delivery of events in the system.

Figure 9: Feral ordering example

```
// Basket microservice: Basket checkout asynchronous request
public async Task CheckoutAsync(Checkout c_out,long userId){
    var basket = await _repository.GetBasketAsync(userId);
    var eventMessage = new CheckoutAcceptedEvent(userId,
         c_out.Buyer, c_out.RequestId, basket);
    try { _eventBus.Publish(eventMessage); }
    catch (Exception ex){
        _logger.LogError(ex, "ERROR Publishing integration
             event: {IntegrationEventId}", eventMessage.Id);
        throw;
    }
}
-------------------------------------------------------------
// Catalog microservice: Update product asynchronous request
public async Task UpdateProductAsync(Item productToUpdate){
    // sets productPriceChanged to true if price changed
         (code omitted)
    if (productPriceChanged){
        var priceChangedEvent = new ProductPriceChangedEvent(
           catalogItem.Id, productToUpdate.Price, oldPrice);
        await _catalogEventService.
           PublishThroughEventBusAsync(priceChangedEvent);
    }
}
-------------------------------------------------------------
// Basket microservice: Reaction to ProductPriceChangedEvent
public async Task Handle(ProductPriceChangedEvent @event){
    var userIds = _repository.GetUsers();
    foreach (var id in userIds){
        var basket = await _repository.GetBasketAsync(id);
        await UpdatePriceInBasketItems(@event.ProductId,
             @event.NewPrice, @event.OldPrice, basket);
    }
}
```

Consider the example adapted from the project eShopContainers [2] shown in Figure 9. The *Basket* microservice receives basket checkout requests through the method *CheckoutAsync*. Additionally, the *Catalog* microservice, upon an item price update request from a user, dispatches a *ProductPriceChanged* event to interested parties. As there is no synchronization, whether the new price will be reflected in the user's order depends on the eventual arrival of events, potentially violating application constraints. The same pattern is found in other projects [2, 18, 37, 52, 69].

Discussion. Literature mentions that developers face challenges in specifying the consistency requirements for their applications [76]. Given that microservices comprise a class of applications that adopt a distributed architectural style [57], it is understandable that defining consistency requirements for these applications may become even more challenging. Indeed, there is a lack of support for developers in reasoning about event-based consistency requirements.

C5. Due to the complex interplay between microservices' behaviors, asynchronous events are generated to trigger computations. However, avoiding anomalies related to the unintended interleaving of events across microservices is a challenging task.

4.6 Replication hell

Background. As mentioned in Section 3.2, microservice architectures may rely on replicating data items across different functional silos to avoid employing synchronous requests spanning multiple microservices for data retrieval and subsequent application-level aggregation. In this context, we observed the prominence of ad-hoc mechanisms for replication. This practice is characterized by the absence of ordering guarantees regarding updates to different objects. The data items arriving from different microservices are often aggregated in queries without any consistency guarantee, i.e., not reflecting a view of the entire system at a single point in time [62].

This pattern is observed in several microservice applications [2, 18, 52, 53, 70, 70].

Discussion. Although one may argue some applications rely on a weak consistency model, such as eventual consistency, to support state querying, it is important to highlight that not all applications fall in this category. Effective mechanisms to support developers are important. In this vein, solutions such as Synapse [78] hold potential to provide more principled replication guarantees in microservices. Even though Synapse [78] is grounded on the industry-strength MVC pattern [24] and meets the polyglot persistence paradigm of microservices (by allowing seamless integration of heterogeneous databases), to the best of our knowledge, the solution has no implementation in frameworks for a variety of programming languages and databases so far, which may hinder a wider industrial adoption. As witnessed in our results, reaching different software development communities is an important feature.

C6. The lack of comprehensive support for data replication in microservice architectures lead developers to rely on ad-hoc application-level replication mechanisms that are difficulty for enforcing consistency semantics across microservices.

5 DELVING INTO DEVELOPERS' PAINS

The investigation presented in the last section revealed a set of microservice data management requirements that are not met by current state of the art database systems. To strengthen our confidence on the practical challenges revealed and derive additional ones, we enquired practitioners about pressing challenges they face while dealing with data management in microservices.

We present next an aggregated discussion over the responses collected from the participants. When appropriate, we include the percentage of each response and the challenges associated. Details about the questions and overall methodology followed in this questionnaire can be found in our extended version [44].

A. Constraint enforcement and schema evolution. In line with **C1** and **C2**, fixing data inconsistencies (19%) and enforcing correctness through application-level validations (24.5%) are prevalent challenges, which, in conjunction with descriptions provided by the respondents, revealed two major interrelated issues.

On the one hand, application-level data consistency and integrity problems, such as "app[lication] code was not [...] removing all usages of an item (in a NoSQL DB[MS])" and "ensur[ing] data is persisted correctly and not duplicated" are mentioned by participants. On the other hand, the use of asynchronous communication for cross-microservice operations or for data replication introduces challenges when it comes to schema evolution in individual microservices. For instance, one respondent declared that "in an async[hronous] environment and having microservices be[ing] responsible for processing their own changes, issues introduced with new releases on microservices may cause inconsistencies in data processing which are generally hard to correct after the fact." Another respondent declared that "as there is no one ruling constraint system, data cleanliness is a significant challenge."

C7. Changes made to a microservice's schema might necessitate adaptations in the structure of messages exchanged; however, application logic in dependent microservices could still be making conflicting assumptions regarding invariants.

B. Eventual consistency. Also prevalent among responses (15.68%), a participant declared: "Dealing with eventual consistency was particularly hard when convergence happens into a broker state. The complexity of the approaches hands a great barrier for developers." We observed two primary drivers for non-converged state in microservices: (i) resorting to event-driven and asynchronous replication to avoid synchronous communication and consequently high latency in online queries (section 4.6); (ii) employing workflow-oriented business transactions across microservices, often driven by asynchronous and event-based communication (Section 3.2).

In both cases, reasoning about the global state of the application is a major concern. In the first case, it is hard to determine when replication has occurred to a sufficient degree, especially as ad-hoc mechanisms are often employed by practitioners. In the second, it is challenging to assert whether a workflow has terminated and what its current state is.

C8. Due to the distributed nature, eventual consistency is often taken as the de facto consistency model by practitioners in microservice architectures. This introduces a series of challenges on reasoning about distributed states and invariants.

C. Ensuring consistency between heterogeneous database systems. Although one may argue that the eventual consistency approach, i.e., convergence through asynchronous events representing data item updates, is a sufficient consistency model for most microservice-based applications, our survey results show that there is a class of applications that require stronger guarantees over writes performed in different database systems (14.70%). For example, a participant argued that "atomicity can not be guaranteed over different storage technologies, no information or proper literature. Guessing and fixing error approach."

In addition, the prevalence of in-memory caching systems for increasing throughput and decreasing data access latency in microservices (Figure 4) may also introduce challenges when developers resort to asynchronous writes. For instance, a respondent declared: "writes are asynchronous to remove the DB[MS] from the critical path [of a data processing pipeline] and ensure that messages can be delivered in near-realtime, [...] but we've already had few situations where the cache expired before the information had actually had time to be persisted in the DB," suggesting measures outside the database were necessary to correct the anomaly.

C9. Developers deal with data inconsistencies with a myriad of ad-hoc measures, such as manual intervention to the underlying microservice databases as well as applying custom-built data management logic to handle possible inconsistencies.

D. Consistent querying. As revealed before (Section 3.2.2 and **C3**), online queries are prominent in microservice architectures. However, there is no de facto approach as developers tend to rely on a variety of ad-hoc mechanisms for online queries (8.82%). Moreover, a consistent view of global state is also expressed as a requirement: "We are integrating data from different sources in a global transportation network. The changes in data are flowing into our system consistently. We need to integrate as fast as possible to present a 'picture of the moment' to the global end-users." To this matter, a respondent indicated "polystore databases [as a solution] to provide location independence and semantic completeness for queries performed by heterogeneous microservices."

E. Poor functional partitioning. Some participants indicated poor functional partitioning as a potential cause of data consistency problems. For instance, one argued that "microservices made people separate code when they should not be separated, causing this eventual consistency everywhere. [...] people wanted to create a separate microservice, just because of 'size', and we end up having consistency problems." Even though initial work suggests considering database-related attributes such as relationships between relations to derive a functional partitioning [43], a thorough evaluation of such approaches with real-world systems requires further exploration.

C10. Decomposing application functionality without proper thought onto constraints and consistency requirements may lead to the burden of dealing with data inconsistencies.

F. Limitations of benchmarks. We realized that existing benchmarks (e.g., DeathStarBench [25]) fail to incorporate the asynchronous and event-driven properties of data-intensive microservices. This can be challenging for programmers to test new solutions.

C11. The lack of a benchmark that properly reflects real-world deployments refrains developers to effectively experiment and reason about microservice deployments.

Summary. In general, the perception of developers regarding the challenges faced while dealing with data management in microservice applications are very aligned with our findings discussed in Section 4. Additional challenges were revealed, such as schema evolution (**C7**), eventual consistency (**C8**), data cleaning (**C9**), and poor functional partitioning (**C10**). This overall perception strengthen our confidence that there is a growing and suppressed need of effective data management support for microservice architectures.

6 IMPLICATIONS FOR DATABASE RESEARCH

The prevalent practice is that microservices are deployed over individual containers, predominantly using the database per microservice pattern in order to achieve performance-and-fault isolation and independent schema evolution, otherwise not met by a shared database. For instance, by resorting to a shared database, microservices are exposed to load spikes and faults caused by buggy-or-slow microservices, compromising the ability of healthy microservices to support their own workload and remain available, respectively.

By pursuing a model of decentralized data management, practitioners end up encountering challenges that should have been solved by database systems. Indeed, the ad-hoc application-level mechanisms observed, and in consequence, the challenges, reveal a myriad of unmet needs. The implications brought by such a design is that databases play a secondary role in the architecture, as they are unaware of the significant amount of data sitting and flowing outside of the database [32]. This makes data management requirements challenging to meet.

D0. Therefore, the **fundamental challenge** is about *system-level abstractions and implementation of data management systems that provide isolation for microservices, which include fault isolation, performance isolation, data ownership, and independent schema evolution across microservices, and at the same time, manage all the data flowing across the architecture.*

Besides, we need to investigate research directions towards meeting the **specific challenges** (sections 4 & 5) that originate from the fundamental challenge. Table 4 summarizes the research directions and their related challenges, explained in more detail below.

D1. Microservice developers have no viable nor efficient solutions to ensure appropriate isolation levels across microservices. The prevalence of cross-microservice validations (Section 4.1) raise the need for more comprehensive database system solutions that meet the programming abstractions that practitioners are more used to work with.

D2. Developers have to explicitly enforce constraints across microservices (Section 4.2), which is a complicated and error-prone task. Support for referential constraint definitions with support for delete rules, such as cascade delete, would refrain developers from handling constraint enforcement through ad-hoc measures.

D3. Programmers often encode data processing logic in the application-tier to query microservices' private states (Section 4.3). As a result, guarantees provided by a database, such as queries with transactional consistency [62], are either no longer possible or hardly achieved without a complex and error-prone implementation. The investigation of principled approaches for online queries in microservices constitutes a fruitful area for further research.

D4. Database systems only offer abstractions where different microservices are forced to access the same database instance for transactional queuing (Section 4.4), which contrasts with the principle of decentralized data management and loose-coupling in microservices [21]. Providing queuing capabilities with transactional guarantees respecting the decentralized data management principle is an important avenue of future work.

D5. The complex interplay of microservices lead developers to deal with possible interleaving of event streams (Section 4.5). Event-based constraints enforcement, including an abstraction for specifying application constraints, such as event processing order and causal consistency, and algorithms and systems for enforcing them, are worthwhile avenues to pursue to fill this observed gap.

D6. Data replication across microservices is a fundamental data management requirement. To alleviate the burden on handling data replication through ad-hoc mechanisms in the application-tier (Section 4.6), an interesting direction includes specifying various

Table 4: Mapping of challenges to research directions

D#	C#	Description
D0	All	Proper database architecture for supporting both functional and performance isolation requirements required by microservices
D1	C{1,6,8}	A programming abstraction that supports isolation levels appropriate for underlying computations and meets programmers' needs
D2	C{2,7}	Support for referential constraint definitions across microservices with support for delete rules, such as cascade delete
D3	C{3,6}	System-level support for consistent querying over decentralized microservices' private states, e.g., point-in-time views
D4	C{4,5}	Integrate message brokers with database systems in a consistent way or provide database system queuing capabilities with transactional guarantees while respecting the decentralized management principle
D5	C{4,5}	Event-based constraints enforcement, including an abstraction for specifying application constraints, such as event processing order and causal consistency
D6	C{3,6}	Programming abstraction to provide explicit data replication at the application-level with different consistency levels
D7	C{2,7,9}	Data quality enforcement features such as schema enforcement to ensure that messages and data items propagated across microservices matches its schema and constraints
D8	C{1,8,11}	System-level abstractions to help developers to reason about and enforce constraints in the absence of safety guarantees when resorting to eventual consistency schemes
D9	C{7,9}	Appropriate interfaces for fixing inconsistent data and data cleaning tools for heterogeneous microservices
D10	C{10,11}	Appropriate tools for modeling distributed schemas towards a database-and-workload-aware partitioning
D11	C11	Design appropriate benchmarks to faithfully meet real-world deployments

consistency models agnostically to the programming language and databases utilized by the programmers.

D7. To address schema evolution natively, we envision system abstractions that implement schema enforcement seamlessly across all microservices to ensure that messages and data items propagated across microservices match their schema and constraints. Besides, as commonly found in recent data warehouse systems [48], intuitive interfaces for constraints on data ingestion, including pruning of anomaly records for further analysis, are also compelling features to alleviate manual interventions in the database.

D8. Resorting to eventual consistency creates a barrier for developers to reason about their data. The prominence of orchestration engines in industry settings may indicate the preference for solutions that allow for finer-grained control, debugging, and observability over the operations spanning multiple microservices. It is worthy that future database system embrace these features.

D9. The challenges promote the emergence of data inconsistencies, forcing developers to rely on manual intervention in the microservices' private databases. It is pressing to provide both appropriate interfaces for fixing inconsistent data and data cleaning tools that run apart of the microservices dataflow or concurrently with, but not altering the underlying computation [11].

D10. Splitting application functionality without database-aware constraints may lead to sub-optimal designs and, subsequently, the burden of dealing with data inconsistencies. While previous work is concentrated on software engineering aspects, database-aware partitioning methods and tools for modeling microservices is an important next step.

D11. Existing benchmarks (e.g. DeathStarBench Suite [25]) fail to incorporate many of the challenges identified, such as querying private states and event-based constraint enforcement. Thus, new benchmarks that reflect real-world industry deployments are necessary to properly assess new system proposals and improvements.

7 CONCLUSION

Microservices are becoming increasingly relevant in industry settings. This paper has presented three interrelated studies aimed at characterizing data management in microservices, namely, a literature review, an analysis of open-source repositories, and an expert survey. Taken together, the results of our studies provide several interesting insights.

The architectural shift in data management brought about by microservices and the lack of comprehensive system support lead practitioners to resort to ad-hoc implementations to fulfill the major motivations and encode the computations. These ad-hoc mechanisms, which often rely on application-level validations, lead to problems that cannot be resolved through code refactorings [22], but rather raise the need for more comprehensive database system solutions that meet the functional and performance requirements of applications.

The system model to effectively meet the challenges posed by microservice architectures remains an open question. The state of the practice and the literature covered in this work present a series of unmet needs and requirements that should be appropriately addressed. Given feasible and effective abstractions, database systems can still have a central role to play in microservice architectures. Thus, we conjecture that novel database systems can be developed to provide a similar level of isolation as the prevalent container-based and database-per-microservice deployment but provide much better data management support to address the revealed challenges.

We hope the results drive the reflection of our community towards effectively meeting the needs of this emerging paradigm.

ACKNOWLEDGMENTS

This work was supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie (grant agreement No 801199) and Independent Research Fund Denmark.

We are grateful to the practitioners that collaborated to this work, gently providing their insights. We are particularly grateful to Diogo Souza (Guiabolso), Gabriel Lima (VTEX), Leonardo Gomes (Amadeus), and Pedro Diniz (VTEX) for their valuable comments.

REFERENCES

[1] Microsoft [n.d.]. *Why Orleans Streams?* Microsoft. https://dotnet.github.io/orleans/Documentation/streaming/streams_why.html

[2] .NET Application Architecture Reference Apps. [n.d.]. *eShopOnContainers*. https://github.com/dotnet-architecture/eShopOnContainers

[3] Leonardo Guerreiro Azevedo, Rodrigo da Silva Ferreira, Viviane Torres da Silva, Maximillien de Bayser, Elton F. de S. Soares, and Raphael Melo Thiago. 2019. Geological Data Access on a Polyglot Database Using a Service Architecture. In *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse* (Salvador, Brazil) *(SBCARS '19)*. ACM, New York, NY, USA, 103–112. https://doi.org/10.1145/3357141.3357603

[4] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1327–1342. https://doi.org/10.1145/2723372.2737784

[5] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2016. Scalable atomic visibility with RAMP transactions. *ACM Transactions on Database Systems (TODS)* 41, 3 (2016), 1–45.

[6] Peter Bailis and Ali Ghodsi. 2013. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Queue* 11, 3 (March 2013), 20–32. https://doi.org/10.1145/2460276.2462076

[7] Jonas Boner, Dave Farley, Roland Kuhn, and Martin Thompson. 2019. *The Reactive Manifesto*. Retrieved August 31, 2020 from https://www.reactivemanifesto.org

[8] Phil Calçado. 2015. *The Back-end for Front-end Pattern (BFF)*. Retrieved September 10, 2020 from https://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff.html

[9] Andrés Carrasco, Brent van Bladel, and Serge Demeyer. 2018. Migrating towards Microservices: Migration and Architecture Smells. In *Proceedings of the 2nd International Workshop on Refactoring* (Montpellier, France) *(IWoR 2018)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3242163.3242164

[10] Luiz Carvalho, Alessandro Garcia, Wesley KG Assunção, Rafael de Mello, and Maria Julia de Lima. 2019. Analysis of the criteria adopted in industry to extract microservices. In *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*. IEEE, 22–29.

[11] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[12] Tse-Hsun Chen, Weiyi Shang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2016. Detecting problems in the database access code of large scale systems: an industrial experience report. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 71–80.

[13] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*. 1001–1012.

[14] Michele Ciavotta, Marino Alge, Silvia Menato, Diego Rovere, and Paolo Pedrazzoli. 2017. A microservice-based middleware for the digital factory. *Procedia Manufacturing* 11 (2017), 931–938.

[15] Transaction Processing Performance Council. 2010. *TPC-C Benchmark*. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[16] P. Cruz, H. Astudillo, R. Hilliard, and M. Collado. 2019. Assessing Migration of a 20-Year-Old System to a Micro-Service Platform Using ATAM. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 174–181.

[17] PostgreSQL 12 Documentation. [n.d.]. *NOTIFY*. Retrieved September 10, 2020 from https://www.postgresql.org/docs/current/sql-notify.html

[18] EdwinVW. [n.d.]. *pitstop*. https://github.com/EdwinVW/pitstop

[19] Tamer Eldeeb and Phil Bernstein. 2016. *Transactions for Distributed Actors in the Cloud*. Technical Report MSR-TR-2016-1001.

[20] Martin Fowler. 2011. *PolyglotPersistence*. Retrieved September 20, 2020 from https://martinfowler.com/bliki/PolyglotPersistence.html

[21] Martin Fowler. 2014. *Microservices a definition of this new architectural term*. Retrieved July 7, 2020 from https://martinfowler.com/articles/microservices.html

[22] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., USA.

[23] 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

[24] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

[25] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[26] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *SIGMOD (SIGMOD '87)*, Vol. 16. Issue 3.

[27] Javad Ghofrani and Daniel Lübke. 2018. Challenges of Microservices Architecture: A Survey on the State of the Practice.. In *ZEUS*. 1–8.

[28] Inc. GitHub. [n.d.]. *GitHub*. https://github.com

[29] Jean-Philippe Gouigoux and Dalila Tamzalit. 2017. From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)* (2017), 62–65.

[30] Wilhelm Hasselbring and Guido Steinacker. 2017. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In *IEEE ICSA Workshops*. 243–246.

[31] Pat Helland. 2017. Life beyond distributed transactions. *Commun. ACM* 60, 2 (2017), 46–54. https://doi.org/10.1145/3009826

[32] Pat Helland. 2020. Data on the Outside vs. Data on the Inside, Vol. 18. ACM Queue. Issue 3.

[33] Gregor Hohpe. 2007. Let's have a conversation. *IEEE internet computing* 11, 3 (2007), 78–81.

[34] Mohammad Imranur, Rahman, Sebastiano Panichella, and Davide Taibi. 2019. A curated Dataset of Microservices-Based Systems. arXiv:1909.03249 [cs.SE]

[35] Apache Kafka. [n.d.]. *KAFKA STREAMS*. Retrieved September 16, 2020 from https://kafka.apache.org/documentation/streams/

[36] Asterios Katsifodimos and Marios Fragkoulis. 2019. Operational Stream Processing: Towards Scalable and Consistent Event-Driven Applications.. In *EDBT*. 682–685.

[37] kbastani. [n.d.]. *event-stream-processing-microservices*. https://github.com/kbastani/event-stream-processing-microservices

[38] B. Kitchenham and S Charters. 2007. Guidelines for performing Systematic Literature Reviews in Software Engineering. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.471.

[39] Barbara A Kitchenham and Shari L Pfleeger. 2008. Personal opinion surveys. In *Guide to advanced empirical software engineering*. Springer, 63–92.

[40] Holger Knoche and Wilhelm Hasselbring. 2019. Drivers and Barriers for Microservice Adoption - A Survey among Professionals in Germany. 14 (01 2019), 1–35. https://doi.org/10.18417/emisa.14.1

[41] A. Krylovskiy, M. Jahn, and E. Patti. 2015. Designing a Smart City Internet of Things Platform with Microservice Architecture. In *2015 3rd International Conference on Future Internet of Things and Cloud*. 25–30.

[42] Rodrigo Laigner, Marcos Kalinowski, Pedro Diniz, Leonardo Barros, Carlos Cassino, Melissa Lemos, Darlan Arruda, Sergio Lifschitz, and Yongluan Zhou. 2020. From a Monolithic Big Data System to a Microservices Event-Driven Architecture. In *46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020, Portorož, Slovenia, Aug 26-28*. 1–8.

[43] Rodrigo Laigner, Sérgio Lifschitz, Marcos Kalinowski, Marcus Poggi, and Marcos Antonio Vaz Salles. 2019. Towards a Technique for Extracting Relational Actors from Monolithic Applications. In *Anais do XXXIV Simpósio Brasileiro de Banco de Dados* (Fortaleza). SBC, Porto Alegre, RS, Brasil, 133–144. https://doi.org/10.5753/sbbd.2019.8814

[44] 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 (Extended Version)*. WorkingPaper. https://curis.ku.dk/admin/files/257372099/preprint.pdf

[45] J. Linaker, S. M. Sulaman, R. Maiani de Mello, and M. Höst. 2015. *Guidelines for Conducting Surveys in Software Engineering*. Technical Report. Lund University. [Publisher Information Missing].

[46] J. Lotz, A. Vogelsang, O. Benderius, and C. Berger. 2019. Microservice Architectures for Advanced Driver Assistance Systems: A Case-Study. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 45–52.

[47] Welder Luz, Everton Agilar, Marcos César de Oliveira, Carlos Eduardo R. de Melo, Gustavo Pinto, and Rodrigo Bonifácio. 2018. An Experience Report on the Adoption of Microservices in Three Brazilian Government Institutions. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering* (Sao Carlos, Brazil) *(SBES '18)*. ACM, New York, NY, USA, 32–41. https://doi.org/10.1145/3266237.3266262

[48] R. Xin M. Armbrust, A. Ghodsi and M. Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR*.

[49] Arthur M. Del Esposte, Fabio Kon, Fabio M. Costa, and Nelson Lago. 2017. InterSCity: A Scalable Microservice-Based Open Source Platform for Smart Cities. In *Proceedings of the 6th International Conference on Smart Cities and Green ICT Systems* (Porto, Portugal) *(SMARTGREENS 2017)*. SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT, 35–46. https://doi.org/10.5220/0006306200350046

[50] M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giaretta, S. T. Larsen, and S. Dustdar. 2018. Microservices: Migration of a Mission Critical System. *IEEE Transactions*

on Services Computing (2018), 1–1.
[51] Antonio Messina, Riccardo Rizzo, Pietro Storniolo, Mario Tripiciano, and Alfonso Urso. 2016. The Database-is-the-Service Pattern for Microservice Architectures, Vol. 9832. 223–233. https://doi.org/10.1007/978-3-319-43949-5_18
[52] Microservice-API-Patterns. [n.d.]. *LakesideMutual.* https://github.com/Microservice-API-Patterns/LakesideMutual
[53] microservices patterns. [n.d.]. *ftgo-application.* https://github.com/microservices-patterns/ftgo-application
[54] G. Márquez and H. Astudillo. 2018. Actual Use of Architectural Patterns in Microservices-Based Open Source Projects. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC).* 31–40.
[55] Gastón Márquez, Mónica Villegas, and Hernán Astudillo. 2018. An Empirical Study of Scalability Frameworks in Open Source Microservices-based Systems. In *International Conference of the Chilean Computer Science Society* (Santiago). 1–8. https://doi.org/10.1109/SCCC.2018.8705256
[56] Davide Neri, Jacopo Soldani, Olaf Zimmermann, and Antonio Brogi. 2019. Design principles, architectural smells and refactorings for microservices: a multivocal review. *SICS Software-Intensive Cyber-Physical Systems* 35, 1-2 (Sep 2019), 3–15. https://doi.org/10.1007/s00450-019-00407-8
[57] Sam Newman. 2015. *Building Microservices* (1st ed.). O'Reilly Media, Inc.
[58] Willian Oizumi, Alessandro Garcia, Leonardo da Silva Sousa, Bruno Cafeo, and Yixue Zhao. 2016. Code Anomalies Flock Together: Exploring Code Anomaly Agglomerations for Locating Design Problems. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16).* Association for Computing Machinery, New York, NY, USA, 440–451. https://doi.org/10.1145/2884781.2884868
[59] Patrick O'Neil. 2018. Escrow Transactions. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. https://doi.org/10.1007/978-1-4614-8265-9_150
[60] Andrew Pavlo. 2017. What are we doing with our lives? Nobody cares about our concurrency control research. In *Proceedings of the 2017 ACM International Conference on Management of Data.* 3–3.
[61] Ilaria Pigazzini, Francesca Arcelli Fontana, Valentina Lenarduzzi, and Davide Taibi. 2020. Towards Microservice Smells Detection. In *Proceedings of the 3rd International Conference on Technical Debt* (Seoul, Republic of Korea) *(TechDebt '20).* Association for Computing Machinery, New York, NY, USA, 92–97. https://doi.org/10.1145/3387906.3388625
[62] Dan RK Ports, Austin T Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. 2010. Transactional Consistency and Automatic Management in an Application Data Cache.. In *OSDI*, Vol. 10. 1–15.
[63] Dan Pritchett. 2008. Base: An Acid Alternative. In *File Systems and Storage*, Vol. 6. ACM Queue. Issue 3.
[64] Kun Ren, Jose M Faleiro, and Daniel J Abadi. 2016. Design principles for scaling multi-core oltp under high contention. In *Proceedings of the 2016 International Conference on Management of Data.* 1583–1598.
[65] Chris Richardson. [n.d.]. *API gateway pattern.* Retrieved September, 15 2020 from https://microservices.io/patterns/apigateway.html
[66] D. Richter, M. Konrad, K. Utecht, and A. Polze. 2017. Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C).* 130–137.
[67] Per Runeson and Martin Höst. 2008. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (Dec. 2008), 131–164. https://doi.org/10.1007/s10664-008-9102-8
[68] Mathijs Saey, Joeri De Koster, and Wolfgang De Meuter. 2018. Skitter: A dsl for distributed reactive workflows. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems.* 41–50.
[69] sczyh30. [n.d.]. *vertx-blueprint-microservice.* https://github.com/sczyh30/vertx-blueprint-microservice
[70] Sentilo. [n.d.]. *Sentilo Platform.* https://github.com/sentilo/sentilo
[71] Vivek Shah. 2017. *Exploration of a Vision for Actor Database Systems.* Ph.D. Dissertation. University of Copenhagen, Denmark.
[72] spring petclinic. [n.d.]. *spring-petclinic-microservices.* https://github.com/spring-petclinic/spring-petclinic-microservices
[73] Davide Taibi. 2020. *A curated list of Open Source projects developed with a microservices architectural style.* https://github.com/davidetaibi/Microservices_Project_list
[74] D. Taibi and V. Lenarduzzi. 2018. On the Definition of Microservice Bad Smells. *IEEE Software* 35, 3 (2018), 56–62.
[75] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. 2019. Microservices Anti-Patterns: A Taxonomy.
[76] Andrew S. Tanenbaum and Maarten van Steen. 2016. *Distributed Systems: Principles and Paradigms* (2nd ed.). CreateSpace Independent Publishing Platform.
[77] Panos Vassiliadis. 2009. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining (IJDWM)* 5, 3 (2009), 1–27.
[78] 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
[79] Markos Viggiato, Ricardo Terra, Henrique Rocha, Marco Tulio Valente, and Eduardo Figueiredo. 2018. Microservices in Practice: A Survey Study. *arXiv e-prints*, Article arXiv:1808.04836 (Aug. 2018), arXiv:1808.04836 pages. arXiv:1808.04836 [cs.SE]
[80] Yiwen Wang, Julio Cesar Dos Reis, Kasper Myrtue Borggren, Marcos Antonio Vaz Salles, Claudia Bauzer Medeiros, and Yongluan Zhou. 2019. Modeling and Building IoT Data Platforms with Actor-Oriented Databases.. In *EDBT.*
[81] Weaveworks. [n.d.]. *Sock Shop : A Microservice Demo Application.* https://github.com/microservices-demo/microservices-demo
[82] H. Zhang, S. Li, Z. Jia, C. Zhong, and C. Zhang. 2019. Microservice Architecture in Reality: An Industrial Inquiry. In *2019 IEEE International Conference on Software Architecture (ICSA).* 51–60.
[83] Olaf Zimmermann. 2017. Microservices Tenets. *Comput. Sci.* 32, 3-4 (July 2017), 301–310. https://doi.org/10.1007/s00450-016-0337-0

A DETAILED METHODOLOGY

A exploratory research methodology [67] concerns investigating existing phenomena to uncover problems as well as foment new ideas and hypotheses for research opportunities.

Data management in microservice architectures is not well studied. Thus, microservice architecture fits in the **context** of an exploratory research, whereas data management forms our **unit of analysis**.

We designed three complementary exploratory studies of the state of the practice aimed at providing a comprehensive overview of data management in microservices.

A.1 Literature Review

Since our study is exploratory, we opted to start with the collection of information from the literature about data management in microservices. Along with unveiling how the literature portrays the development of microservices projects, the literature review allowed us to build a foundation to properly explore additional challenges through analyzing open source repositories and defining the survey questions aimed at expert practitioners.

We followed the guidelines of Kitchenham and Charters [38] to select compelling studies for analysis. We targeted studies describing the adoption of microservice architectures, such as through a green-field development (i.e., starting from scratch) or a migration from a legacy system. The second primary criterion was the degree of data management discussion provided by the study. In other words, studies with limited discussion about data management, evidenced for example by lack of information about consistency semantics in the application or technical motivations on choosing specific database technologies, were removed from our analysis, since they would not help us properly study the practice of data management and delve into the challenges practitioners face.

In summary, a total of more than 300 papers retrieved from different digital libraries, such as Scopus and Scholar, were analyzed and a set of 11 papers [3, 14, 16, 30, 41, 42, 49, 50, 66, 78] were able to meet our selection criteria. Even though very few papers provided detailed enough information about data management in microservices, we believe these studies are strongly representative of the peer-reviewed literature in light of our unit of analysis.

A.2 Analysis of Open-source Repositories

Analyzing open-source repositories constitutes an important procedure to understand a phenomenon that occurs in software systems

as well as to validate propositions about such a phenomenon [4, 58]. By analyzing open-source repositories of microservice-based applications, we are able to delve further into how data management schemes are implemented in such an architectural style.

Selection Goal. A primary aspect of the selection of open-source applications is to include a heterogeneous body of strategies for data management in microservices. In other words, we aim at representing a variety of application domains, programming languages and frameworks, and database technologies.

Selection Criteria. Based on our selection goal, we defined a set of quality criteria to identify projects as follows: (i) the project must portray a real-world business scenario even if it is not used in production settings; (ii) the project must exhibit historical developer engagement, i.e., several commits performed by different developers; (iii) the project must enjoy a relatively high popularity, e.g., in terms of stars; (iv) the application must have a significant number of microservices; (v) the application must present characteristics that allow us to characterize data management challenges, such as exhibiting substantial data management logic in the microservices.

Search Process. Since microservices are an emerging trend, the availability of microservice open-source repositories is not substantial in comparison to other classes of applications. Thus, we relied on catalogued data sets of open-source repositories [34, 54, 55, 73] and exploratory searches on GitHub [28]. We analyzed the characteristics, such as source code, metrics, and popularity, of more than 20 projects. In total, 9 open-source projects [2, 18, 37, 52, 53, 69, 70, 72, 81] were selected for further analysis.

A.3 Expert Survey

The first two steps of our research methodology are designed to provide us with a foundation on the topic, i.e., acknowledge the main terms and technologies applied in microservice architectures. To complement and validate the findings derived from these two steps, we designed an exploratory survey [45]. Here, the primary goal is to further detail the state of the practice of microservices concerning data management as well as to identify emerging trends among practitioners and researchers. It is worthy to note that in the design of our survey, we adopted the guidelines of Kitchenham and Pfleeger [39].

Population. Regarding the target population, we aimed to select industrial and academic representatives in a way that is most appropriate to achieve our goal, i.e., by focusing on those with a background on the development or maintenance of microservice-based systems in real-world settings.

Dissemination Strategy. We pursued a wide dissemination strategy, targeting developers and researchers working with microservices with diverse backgrounds and from different companies and communities. We considered multiple methods to recruit participants, such as direct emails, LinkedIn, Twitter, Facebook groups, Google groups, Slack Channels, and discussion lists (e.g., meetup groups). It is worthy to note that we primarily approached microservice development groups and communities on the web.

Instrumentation. We designed an online questionnaire and there were 3 types of questions: (i) single-choice questions; (ii) multiple-choice questions; and (iii) open questions, where respondents could enter their responses freely. For both single- and multiple-choice questions, the participants could enter an *Other* option if the answer did not match any of the provided options or further explanation was required. Besides, for single- and multiple-choice questions, we alphabetically sorted the order of the options, except for the *Other* option that was always the last one.

There were 27 questions grouped into 3 high-level categories: (i) background information about the participants; (ii) state of the practice of data management in microservices (e.g., databases, types of communication, consistency semantics, and application invariant validations); and (iii) challenges faced. A total of 123 responses were collected from July 20th to September 14th, with an average completion rate of 50%. We give an overview of the experience, the sizes of organizations, and the roles of the participants as follows.

The results obtained through the three steps of our methodology are jointly discussed in the next sections, which are organized by the main observations emerging from our study.

Table 5: Engines for inter-microservice operations

Orchestration engine	#	%
Custom-made (e.g., company-built engine)	20	51.28
Spring Integration	4	10.25
Apache Camel	4	10.25
Netflix Conductor	2	5.12
Apache Airflow	2	5.12
Zeebe	1	2.56
Others	6	15.38

B STATE OF THE PRACTICE

B.1 Orchestration engines used for inter-microservice operations

Table 5 shows the responses.

B.2 Types of Communication

Table 6 shows the responses.

We asked the participants to characterize the type of message patterns employed to ensure data consistency across microservices. By digging deeper into the responses and open-source repositories, we observe the following patterns:

(i) Persistent, asynchronous message-passing is the most used type of communication (42% of respondents) in cases where an operation spans multiple microservices. The fault tolerance provided by a middleware service (e.g., message broker) is essential to safeguard that a given request will eventually reach every microservice.

(ii) Synchronous communication (39.5% of respondents), mostly through REST APIs, is often used to serve client requests, such as queries or updates to data items. In other words, REST APIs target interactive functionalities, characterized by client-oriented, synchronous requests.

These results suggest that data management approaches for microservices should take into account these dominant types of communication found in practice.

Table 6: Messaging patterns for microservices

Messaging pattern	#	%
Asynchronous requests enabled by a message broker (e.g., RabbitMQ, ActiveMQ, or Kafka)	50	42%
Synchronous requests via HTTP protocol (e.g., RESTful APIs or GraphQL)	47	39.5%
Synchronous requests via Remote Procedure Calls (e.g., gRPC or Apache Dubbo)	11	9.25%
Asynchronous requests enabled by programming language or framework (e.g., Elixir, Erlang messages)	8	6.75%
Other	3	2.5%

Table 7: Database design patterns for microservices

Database pattern	#	%
Database per microservice (i.e., a database server per microservice)	34	43.58
Schema per microservice (i.e., a schema per microservice, sharing a database server)	26	33.33
Private tables per microservice (i.e., a set of tables per microservice, sharing a schema and a database server)	13	16.66
Others	5	6.41

B.3 Database Patterns

There are three mainstream approaches for using database systems in microservice architectures: (i) private tables per microservice, sharing a database server and schema; (ii) schema per microservice, sharing a common database server; and (iii) database server per microservice [51].

We asked the participants which database patterns they have been using to support data management in their microservices and the responses are shown in Table 7. We also asked the participants what drivers led to the adoption of each database pattern in their microservice architectures. Due to space limitations, we comment on those below without showing the survey responses.

The prevalence of a database per microservice: The results show that more practitioners prefer encapsulating a microservice state within its own managed database server. The same trend is observed in the literature and open-source repositories.

The participants indicated that achieving loosely-coupled microservices is the most important driver for adopting this pattern. The second most cited driver is the ability to scale each microservice's underlying database independently, which would otherwise be challenging with a single database server supporting multiple (heterogeneous) tenant applications. The third most cited motivation for this pattern is fault-isolation, which is naturally derived from the already mentioned formation of independent silos of data.

The popularity of a schema per microservice: We also observed a high number of participants indicating the use of a schema per microservice pattern. Again, loose coupling is the most mentioned driver for this pattern. Although isolation of faults is not possible as in the database per service pattern, a schema per microservices still enables the establishment of clear boundaries between microservices. Next, decreased maintenance efforts and agility to prototype a system were also highly mentioned (15 and 14 responses, respectively). Interestingly, some participants (10) indicated the use of schema per microservice as an initial boundary for future adoption of the database per service pattern.

The unpopularity of private tables per microservice: Only 13 practitioners indicated using the private tables per microservice pattern. Practitioners indicated that agility to prototype a system and decreased maintenance efforts were the most important drivers for the adoption of such a pattern (4 and 3, respectively).

Table 8: Challenges for ensuring data consistency

Challenge	#	%
Fixing data inconsistencies caused by application-level validations (e.g., manually intervention to database)	39	19.11
Eventual consistency, a weak consistency model (i.e., hard to determine when state will actually converge)	32	15.68
Ensuring consistency between heterogeneous database systems	30	14.70
The use of application-level validations to ensure data integrity	25	12.25
Interleaved requests on individual microservices, which may impose race conditions, thus leading to data integrity problems	25	12.25
Lack of interoperability among heterogeneous databases	18	8.82
The lack of proper discussion on literature about consistency mechanisms for microservices	17	8.33
Insufficient programming language or framework support	11	5.38
Others	7	3.44

C CHALLENGES IN PRACTICE

In this section, we discuss challenges associated to data management in microservices. These challenges were identified by cross-validating the participants' opinions with issues identified through the literature review and analysis of open-source repositories.

We start by presenting the results of a question where we asked the participants to indicate the top-3 most pressing challenges for ensuring data consistency in microservice architectures. We chose this question because existing studies [23, 40, 47, 50, 79, 82] argue that achieving consistency is a major issue in microservices. The options were defined based on related challenges reported in the literature [4, 6] and on a preliminary assessment of the survey with pre-selected participants, and further evolved through discussions among the authors. Besides, through the *Other* option, participants could freely identify additional challenges.

Table 8 shows the options and respective responses. We also asked the participants to briefly describe at least one of the challenges faced in the context of a project they have been involved in and to indicate opportunities to improve data management in microservices. We received 35 descriptions from both questions in total and discuss the key challenges identified as follows.

Constraint enforcement and schema evolution. Fixing data inconsistencies and enforcing application-level validations are prevalent challenges that in conjunction with descriptions provided by the respondents, revealed two major interrelated issues.

On the one hand, application-level data consistency and integrity problems, such as "app[lication] code was not [...] removing all usages of an item (in a NoSQL DB[MS])" and "ensur[ing] data is persisted correctly and not duplicated" are mentioned by participants. This result suggests that findings from previous work targeting monolithic web applications may also apply to this context [4].

On the other hand, the use of asynchronous communication for cross-microservice operations or for data replication introduces challenges when it comes to schema evolution in individual microservices. For instance, one respondent declared that "in an async[hronous] environment and having microservices be[ing] responsible for processing their own changes, issues introduced with new releases on microservices may cause inconsistencies in data processing which are generally hard to correct after the fact." The participant further declares that the absence of "a historical transaction log of async[hronous] messages between [micro]services" is an impedance to properly identifying when integrity constraints are violated.

In other words, changes made to a microservice's schema might necessitate adaptations in the structure of messages exchanged; however, application logic in dependent microservices could still be making conflicting assumptions regarding invariants [32]. For instance, a respondent declared that "as there is no one ruling constraint system, data cleanliness is a significant challenge." The person carries on by arguing that "identifying old valid date [sic] that is no longer valid in a petabyte dataset is hard." Another respondent declared that "manual compensation by developers or customer support" is a common procedure. This finding suggests that additional research efforts are necessary to effectively support information integration, constraint enforcement, and data cleaning in microservices.

Lastly, some participants indicated poor functional partitioning as a potential cause of data consistency problems. For instance, one argued that "microservices made people separate code when they should not be separated, causing this eventual consistency everywhere. [...] people wanted to create a separate microservice, just because of 'size', and we end up having consistency problems." Even though initial work suggests considering database-related attributes such as relationships between relations to derive a functional partitioning [43], a thorough evaluation of such approaches with real-world systems requires further exploration.

Eventual consistency. A participant declared: "Dealing with eventual consistency was particularly hard when convergence happens into a broker state. The complexity of the approaches hands a great barrier for developers." We observed two primary drivers for non-converged state in microservices: (i) resorting to event-driven and asynchronous replication to avoid synchronous communication and consequently high latency in online queries; (ii) employing workflow-oriented business transactions across microservices, often driven by event-driven and asynchronous communication.

In both cases, reasoning about the global state of the application is a major concern. In the first case, it is hard to determine when replication has occurred to a sufficient degree, especially as ad-hoc mechanisms are often employed by practitioners. In the second, it is challenging to assert whether a workflow has terminated and what its current state is. Therefore, the prominence of orchestration engines in industry settings may indicate the preference for solutions that allow for finer-grained control, debugging, and observability over the operations spanning multiple microservices.

We believe there is an open area for development of database systems that embrace these new business transaction patterns. One possibility would be to employ database systems as orchestration frameworks, providing the desired capabilities along with data consistency semantics adequate for the requirements of a data-driven application.

Ensuring consistency between heterogeneous database systems. Although one may argue that the eventual consistency approach, i.e., convergence through asynchronous events representing data item updates, is a sufficient consistency model for most microservice-based applications, our survey results show that there is a class of applications that requires stronger guarantees over writes performed in different database systems. For example, a participant argued that "atomicity can not be guaranteed over different storage technologies, no information or proper literature. Guessing and fixing error approach."

In addition, the prevalence of in-memory caching systems for increasing throughput and decreasing data access latency in microservices (Figure 4) may also introduce challenges when developers resort to asynchronous writes. For instance, a respondent declared: "writes are asynchronous to remove the DB[MS] from the critical path [of a data processing pipeline] and ensure that messages can be delivered in near-realtime, [...] but we've already had few situations where the cache expired before the information had actually had time to be persisted in the DB."

The efforts to formalize standards for transactions across database resources failed to experience wide adoption among practitioners [32]. Despite important advances [12, 13, 62], it remains open how to best provide methodologies to achieve data consistency across heterogeneous data management systems while minimizing coupling and providing compelling interfaces for developers.

Other challenges. In the survey results, 11 participants mentioned that the lack of proper discussion in the literature about data consistency mechanisms in microservices is a significant challenge. Furthermore, insufficient programming language or framework support was indicated by 11 participants.

In addition, as revealed by our survey results, online queries are prominent in microservice architectures. However, there is no de facto approach as developers tend to rely on a variety of ad-hoc mechanisms for online queries. Moreover, a consistent view of global state is also expressed as a requirement: "In our project, we are integrating data from different sources in a global transportation network. The changes in data are flowing into our system consistently. We need to integrate as fast as possible to present a 'picture of the moment' to the global end-users." To this matter, a respondent indicated "polystore databases [as a solution] to provide location independence and semantic completeness for queries performed by heterogeneous microservices." The investigation of principled approaches for online queries in microservices constitutes a fruitful area for further research.

Limitations of benchmarks. Although at first we considered incorporating the projects described in the DeathStarBench Suite [25]

in our open-source repository analysis, we opted to include only open-source projects built and maintained by open-source contributors. Later on, we realized that the DeathStarBench Suite [25] mainly targets REST and RPC calls, not incorporating the event-driven and asynchronous communication properties of microservices that we uncovered.

In addition, existing benchmarks for business transactions in database research (e.g., TPC-C [15]) target online transaction processing (OLTP) workloads, characterized by short, non-interactive transactions [64]. In contrast with OLTP, we unveiled that business transactions across microservices are characterized by an event-driven, workflow-oriented composition of tasks performed by different microservices.

Thus, new benchmarks for data management in microservices that reflect real-world industry deployments are necessary to properly assess new system proposals and improvements.