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

Publication date:
2021

Document version
Early version, also known as pre-print

Document license:
CC BY-NC-ND

Citation for published version (APA):
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)).
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 discuss 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
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.

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...
We were also interested in characterizing the types of queries and computations identified from the responses, characterized by Cross-Microservice Business Transactions (BT), Event-Driven Computing (ED), Online Queries (OQ), and Stream Processing (SP).

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

---

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.

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).

---

Figure 3: Computations indicated by participants
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. In this mechanism, a service centralizes the role of performing requests across microservices.

### Table 3: Mechanisms for inter-microservice coordination

<table>
<thead>
<tr>
<th>Coordination mechanism</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orchestration</td>
<td>37</td>
</tr>
<tr>
<td>Sagas (centralized approach, with a Saga coordinator)</td>
<td>24</td>
</tr>
<tr>
<td>The Back-end for Front-end Pattern (BFF)</td>
<td>24</td>
</tr>
<tr>
<td>Choreography</td>
<td>22</td>
</tr>
<tr>
<td>Sagas (decentralized approach, i.e., no Saga coordinator)</td>
<td>14</td>
</tr>
<tr>
<td>Distributed transactions (e.g., via 2PC)</td>
<td>14</td>
</tr>
<tr>
<td>2-Phase Commit</td>
<td>11</td>
</tr>
<tr>
<td>Others</td>
<td>16</td>
</tr>
</tbody>
</table>

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.

**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.

**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 dynamics provided by the autonomous deployment of microservices plays a role.

---

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.
We use Kafka Streams to reorder (time window) out-of-sequence events. 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 Data Stream Processing Systems (DSPSs) and 12 provided a short answer. 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 on 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.
3.4 Database Systems

We asked the participants what database systems they have been using 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 crashing, 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. 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 complex 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.
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 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.

**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...
After adding several items to a cart, which is managed by the delivery of events in the system. Constraints are such that price changes should not affect such orders. This requires that events related to changes in the price of items should be considered in the context of the customer's cart. Event representing the change of the price of one of the items in the cart is an example of an event exchange between microservices. Without interoperability in the form of databases and message brokers, 2PC is not possible, violating a critical aspect of databases and message brokers for the communication and event exchange between microservices. As a result, most open-source projects and the literature rely on external message brokers for the communication and event exchange between microservices.

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 store for the data, they do not inherently support the transactional properties required for the applications shown. 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 8: Non-transactional queuing example**

```java
public ResponseEntity createInsuranceQuote() {
    InsuranceQuote insuranceQuote = new InsuranceQuote
    (date, QUOTE_SUBMITTED, insuranceQuoteOptionsEntity);
    insuranceQuoteRepository.save(insuranceQuote);
    return ResponseEntity.ok(insuranceQuote);
} // parameters omitted
```

```java
public void send(Date date, InsuranceQuote insuranceQuote) {
    InsuranceQuoteEvent insuranceQuoteEvent =
    new InsuranceQuoteEvent(date, insuranceQuote);
    try { jmsTemplate.send(insuranceQuoteEventQueue,
    new InsuranceQuoteEvent); }
    catch (JmsException exception) {
        logger.error("Failed to send", exception);
    }
} // parameters omitted
```

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 store for the data, they do not inherently support the transactional properties required for the applications shown. 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**

```java
// Basket microservice: Basket checkout asynchronous request
public async Task CheckoutAsync(long userId){
    var basket = await _repository.GetBasketAsync(userId);
    var eventMessage = new BasketAcceptedEvent(userId, basket);
    try { _eventBus.Publish(eventMessage); }
    catch (Exception ex){
        _logger.LogError(ex, "ERROR Publishing integration event: {IntegrationEventId}", eventMessage.Id);
        throw;
    }
} // parameters omitted
```

```java
// Catalog microservice: Update product asynchronous request
public async Task UpdateProductAsync(Item productToUpdate){
    // sets productPriceChanged to true if price changed
    if (productPriceChanged){
        var priceChangedEvent = new ProductPriceChangedEvent(
        catalogItem.Id, productToUpdate.Price, oldPrice);
        await _catalogEventService.
        PublishThroughEventBusAsync(priceChangedEvent);
    }
} // parameters omitted
```

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 difficult 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 “application code was not [...] removing all usages of an item (in a NoSQL DB[MS])” and “ensuring 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 asyncronous 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.
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.

D6. 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

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

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 worth noting 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 (Guiabolsco), Gabriel Lima (VTEX), Leonardo Gomes (Amadeus), and Pedro Diniz (VTEX) for their valuable comments.
A DETAILED METHODOLOGY

An 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.

### B STATE OF THE PRACTICE

#### B.1 Orchestration engines used for inter-microservice operations

Table 5 shows the responses.

<table>
<thead>
<tr>
<th>Orchestration engine</th>
<th>#</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom-made (e.g., company-built engine)</td>
<td>20</td>
<td>51.28</td>
</tr>
<tr>
<td>Spring Integration</td>
<td>4</td>
<td>10.25</td>
</tr>
<tr>
<td>Apache Camel</td>
<td>4</td>
<td>10.25</td>
</tr>
<tr>
<td>Netflix Conductor</td>
<td>2</td>
<td>5.12</td>
</tr>
<tr>
<td>Apache Airflow</td>
<td>2</td>
<td>5.12</td>
</tr>
<tr>
<td>Zeebe</td>
<td>1</td>
<td>2.56</td>
</tr>
<tr>
<td>Others</td>
<td>6</td>
<td>15.38</td>
</tr>
</tbody>
</table>

#### 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.**
### 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).

### C CHALLENGES IN PRACTICE

In this section, we discuss challenges associated to data management in microservices. These challenges were identified by crossvalidating 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.

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

#### Table 7: Database design patterns for microservices

<table>
<thead>
<tr>
<th>Database pattern</th>
<th>#</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database per microservice (i.e., a database server per microservice)</td>
<td>34</td>
<td>43.58</td>
</tr>
<tr>
<td>Schema per microservice (i.e., a schema per microservice, sharing a database server)</td>
<td>26</td>
<td>33.33</td>
</tr>
<tr>
<td>Private tables per microservice (i.e., a set of tables per microservice, sharing a schema and a database server)</td>
<td>13</td>
<td>16.66</td>
</tr>
<tr>
<td>Others</td>
<td>5</td>
<td>6.41</td>
</tr>
</tbody>
</table>

#### Table 6: Messaging patterns for microservices

<table>
<thead>
<tr>
<th>Messaging pattern</th>
<th>#</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous requests enabled by a message broker (e.g., RabbitMQ, ActiveMQ, or Kafka)</td>
<td>50</td>
<td>42%</td>
</tr>
<tr>
<td>Synchronous requests via HTTP protocol (e.g., RESTful APIs or GraphQL)</td>
<td>47</td>
<td>39.5%</td>
</tr>
<tr>
<td>Synchronous requests via Remote Procedure Calls (e.g., gRPC or Apache Dubbo)</td>
<td>11</td>
<td>9.25%</td>
</tr>
<tr>
<td>Asynchronous requests enabled by programming language or framework (e.g., Elixir, Erlang messages)</td>
<td>8</td>
<td>6.75%</td>
</tr>
<tr>
<td>Other</td>
<td>3</td>
<td>2.5%</td>
</tr>
</tbody>
</table>
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 "application code was not [...] removing all usages of an item (in a NoSQL DB[MS])" and "ensuring 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 asynchronous environment and having microservices being 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 asynchronous 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.