UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE

Ph.D. Thesis

Rodrigo Nunes Laigner

A Scalable and Consistent Database System for Event-driven Architectures

Advisor: Dr. Yongluan Zhou

This thesis has been submitted to the Ph.D. School of The Faculty of Science, University of Copenhagen on 28/02/2025.

Abstract

Cloud computing has promoted a profound transformation in how computing services are acquired, maintained, billed, and deprecated. By enabling the pay-as-you-go model, that is, the on-demand utilization of computational resources without the burden of managing hardware infrastructure, cloud computing has prompted service providers and application maintainers to rethink the design of software systems to better adhere to this computational resource rental model.

In particular, we have recently witnessed an increasing adoption of event-driven and microservice architectures in the industry. These architectures promote a highly modular software design and are characterized by deploying small and independent components that communicate via asynchronous messages, sharing no state or computational resources. Such a design in the cloud ameliorates the provisioning and specialization of computational resources holistically, adapting to the needs of each component, and prevents failures from propagating outside the boundary of a component.

Despite the accelerated industrial adoption and relevant evidence that data management is a major challenge in event-driven and microservice architectures, there has been a lack of thorough investigation into the state of the practice and the major challenges faced by practitioners with regard to data management in such an architectural style, which jeopardizes the advancement of data management solutions to effectively meet practitioners' needs.

To fill this gap, this dissertation starts by presenting an in-depth investigation of the state of the practice of data management in microservices, revealing several unmet data management requirements and foundational challenges that require rethinking the design and architecture of database management systems (DBMSs).

To aid this development, this research proposes *Online Marketplace*, a novel microservice benchmark that prescribes key data management requirements that existing benchmarks fail to incorporate, including transaction processing, query processing, event processing, constraint enforcement, and data replication, allowing for properly comparing data systems and platforms. Throughout the *Online Marketplace* implementation and evaluation in competing systems, we found a mismatch between data management requirements and state-of-the-art data platforms for cloud applications, re-

quiring weaving together heterogeneous systems to achieve the requirements fully.

Finally, this dissertation answers the long-standing question of whether event-driven architectures (EDAs) are fated to eventual consistency. We propose the virtual Microservice Oriented DataBase (vMODB), a novel distributed DBMS that tackles the problem of data inconsistency in distributed and asynchronous applications by design. We propose the virtual micro service (VMS) programming model, through which vMODB leverages application semantics to enforce ACID properties. vMODB transparently unifies event and data management into a common event-driven execution framework, all without giving up the envisioned benefits of EDAs. As a result, vMODB will significantly simplify the design and implementation of highly consistent applications based on EDA.

Resumé

Skybaseret databehandling har fremmet en dybtgående transformation i, hvordan databehandlingstjenester erhverves, vedligeholdes, faktureres og udfases. Ved at muliggøre pay-as-you-go-modellen, dvs. den efterspørgselsbaserede anvendelse af computerressourcer uden byrden af at administrere hardwareinfrastruktur, har skybaseret databehandling fået tjenesteudbydere og applikationsvedligeholdere til at gentænke designet af softwaresystemer for bedre at overholde denne model for leje af computerressourcer.

Især har vi for nylig været vidne til en stigende adoption af hændelsesdrevne og mikroservice-arkitekturer i industrien. Disse arkitekturer fremmer et meget modulært softwaredesign og er kendetegnet ved at implementere små og uafhængige komponenter, der kommunikerer via asynkrone beskeder, uden at dele tilstand eller computerressourcer. Et sådant design i skyen forbedrer holistisk tildelingen og specialiseringen af computerressourcer, tilpasser sig behovene for hver komponent og forhindrer fejl i at sprede sig uden for en komponents grænser.

På trods af den accelererede industrielle adoption og relevant evidens for, at datastyring er en stor udfordring i hændelsesdrevne og mikroservicearkitekturer, har der været en mangel på grundig undersøgelse af praksisens tilstand og de store udfordringer, som praktikere står over for med hensyn til datastyring i en sådan arkitektonisk stil, hvilket bringer udviklingen af datastyringsløsninger i fare for effektivt at imødekomme praktikernes behov.

For at udfylde dette hul præsenterer denne afhandling en dybdegående undersøgelse af praksisens tilstand for datastyring i mikroservices, der afslører flere uopfyldte datastyringskrav og grundlæggende udfordringer, der kræver en gentænkning af design og arkitektur af database management systemer (DBMS'er).

For at støtte denne udvikling foreslår denne forskning *Online Marketplace*, en ny mikroservice-benchmark, der foreskriver nøglekrav til datastyring, som eksisterende benchmarks ikke formår at inkorporere, herunder transaktionsbehandling, forespørgselsbehandling, hændelsesbehandling, håndhævelse af begrænsninger og datareplikering, hvilket muliggør korrekt sammenligning af datasystemer og platforme. Gennem implementeringen og evalueringen af *Online Marketplace* i konkurrerende systemer fandt vi en uoverensstemmelse mellem datastyringskrav og state-of-the-art dataplatforme for skyapplikationer, hvilket kræver sammenvævning af heterogene systemer for fuldt ud

at opfylde kravene.

Endelig besvarer denne afhandling det langvarige spørgsmål om, hvorvidt hændelsesdrevne arkitekturer (EDA'er) er skæbnebestemt til eventual consistency. Vi foreslår den virtuelle Microservice Oriented DataBase (vMODB), en ny distribueret DBMS, der tackler problemet med datainkonsistens i distribuerede og asynkrone applikationer ved design. Vi foreslår den virtuelle mikroservice (VMS) programmeringsmodel, gennem hvilken vMODB udnytter applikationssemantik til at håndhæve ACID-egenskaber. vMODB forener transparent hændelses- og datastyring i en fælles hændelsesdrevet eksekveringsramme, alt sammen uden at opgive de forudsete fordele ved EDA'er. Som et resultat vil vMODB betydeligt forenkle design og implementering af meget konsistente applikationer baseret på EDA.

Acknowledgements

I dedicate this thesis to my sweet Jesus Christ, through whom I shall be saved, for His salvific sacrifice and accompanying me in the darkest hours along this journey. I also dedicate this work to my lovely godmother, Maria Jose Teixeira, and grandfather, Jose Apolinario, who passed away during this journey. You will always be remembered in my heart. Finally, I dedicate this work to my family, particularly my mother and sister, who supported me and understood the sacrifices required to achieve my dreams. A special thanks to Eider for being such a special person and caring for me.

I do not have enough words to express the gratitude, respect, and admiration that I have for my advisor, Prof. Yongluan Zhou. His wisdom, leadership, and encouragement were key to reaching this level. Thanks for guiding me throughout the hardships of this process and believing in my potential. I am also very grateful to my former co-advisor, Marcos Antonio Vaz Salles, who also believed in my potential and encouraged me to apply for a PhD at KU. His intelligence, humbleness, and our research discussions were like lamps to my days. I would also like to express my sincere gratitude to Prof. Asterios Katsifodimos and his research group for welcoming me into my exchange of environment. Our discussions were fulfilling and form part of this thesis. I extend my gratitude to Panagiotis Karras, Zoi Kaoudi, and Paris Carbone for kindly agreeing to serve on the assessment committee.

A special thanks to all my other co-authors, Yijian Liu, Zhexiang Zhang, Leonardo Gomes, and Marcos Kalinowski. I would like to express my gratitude to the SPDS Section, DIKU, the University of Copenhagen, and the many extraordinary people that I met along this journey, in particular Yijian, Tilman, Li, Yibin, and Xikun, for our friendship in the lab.

I especially thank the friends who welcomed me along this journey. I thank Cida and family, and also Michael, for their kindness and welcome. I thank Luiz Assis, Renilce, Tulio, and Rafael for the welcome and care. I thank Fabrizio, Fon, my cousin Marcelo and his spouse, Thales, and Erick. I thank the many friends I have made, in particular, David, Oleg, Lorenzo, Zafer, Alfifi, Rodrigo, Chaewon, Jean, Plinio, and Rohit.

This PhD work was supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie agreement No 801199 and Independent Research Fund Denmark grant No 9041-00368B.

Contents

Chapter 1

Introduction

1.1 Motivation

The emergence of cloud computing as a key paradigm for software and infrastructure as a service has prompted decision-makers and software teams in organizations to rethink their strategies for designing, implementing, deploying, monitoring, deprecating, and upgrading their applications. In the cloud service paradigm, cloud providers acquire and maintain computational resources at scale, such as data centers and servers, enabling the pay-per-use model, on which users pay solely for the resource utilized and the time consumed, relieving users from dealing with computing infrastructure concerns. To better fit the on-demand resource provisioning model, software systems are increasingly being designed following a modular approach so to manage each module as an independent component in the cloud. Modules can be assigned specialized resources for their computational tasks, accounting for their individual resource constraints and performance requirements, allowing for higher cost-efficient cloud deployments.

Microservice-based architectures arose as a prominent architectural style for scalable and distributed applications in the cloud. These architectures are characterized by a design where an application is composed of self-contained, autonomous components, each encapsulating its own private state. Each component represents a specific and distinct functionality, for example, stock management and payment processing, and can be evolved and torn down without affecting the whole application. Although a component operates independently, it may be composed to realize a business transaction. For

example, placing an order would involve withdrawing some items from the stock and processing the corresponding payment details. By promoting a model where components share no state while the communication overhead remains minimal, these interactions are often performed via asynchronous events, message payloads that correspond to data, or operations that ought to be processed by a receiver component. To better support partial failures, events are persisted in message middlewares (e.g., message queues like RabbitMQ or event log systems like Kafka), allowing events to be decoupled in time, characterizing then the event-driven microservice architecture.

From a technical perspective, the benefits include the fine-grained and holistic provision of computational resources, scaling performance for specific microservices and clients, faster reaction to dynamic workloads, quicker upgrades and bug fixing, easier management of code repositories, facilitation of feature and service reuse, integration with external and legacy systems, data decentralization, and event-driven data management.

From an organizational point of view, it boosts the independence of development teams since it allows different teams to implement and evolve their microservices with little coordination with others. As a result, microservices and message types can be easily added to and demoted from the application without service disruption. Similarly, it facilitates the introduction of new tenants, promoting fast-paced changes and supporting business innovation.

The popularization of microservice architectures has prompted cloud providers to offer rich features targeting microservice deployments [49, 16, 50, 13], such as specialized container-based technologies to deploy and scale microservices, message brokers to allow for loose-coupled microservices, multi-tenant database management system (DBMS) technologies to meet the isolation properties of microservices, and application frameworks and side-car technologies to speed up deployment, code evolution and maintenance of microservices. For instance, Amazon AWS asserts that services such as EventBridge, SNS, SQS, and Step Functions facilitate building event-driven microservice architectures [12].

However, apart from the popularity of microservice architectures in industry settings and their importance in the modern cloud application development landscape [235], there is little research about microservice architectures from a data management perspective. More importantly, although there is widespread evidence positioning data integrity as a major challenge in microservice architectures [134, 261, 94, 162, 278, 227], the characteristics of data management in such architectures have not been thoroughly studied.

As a result, it is unclear how to advance data management solutions for this important class of applications.

In this thesis, we identify four fundamental gaps:

Gap #1: Little is known about how microservice developers encode data management logic and interact with database systems, which data integrity constraints and workloads are pursued, which database systems and patterns are adopted, and which mechanisms are used to exchange data across components.

Gap #2: There is a lack of a comprehensive benchmark incorporating and measuring the performance of data management tasks in the microservice architectural style, making it difficult to compare and evaluate different data system technologies for supporting microservice applications.

Gap #3: Despite the recent proliferation of solutions that facilitate the development of cloud applications, little is known about whether and how these competing cloud programming paradigms and systems differ, as well as their provided semantics and key limitations, making it difficult to characterize challenges related to database research.

Gap #4: Existing programming models of data system technologies fail to fulfill the data management requirements of microservice applications, burdening application developers with enforcing data integrity. Consequently, it is unclear whether it is possible to devise a database architecture to natively support microservice data management requirements in database systems, which can relieve the burden on microservice developers.

1.2 Contributions

To fill the aforementioned gaps, this thesis makes the following contributions:

Contribution #1. We start with designing a large-scale survey on the state of the practice and challenges in data management in the popular microservices architectural style.

The prevalent practice is that microservices are implemented via (possibly different) high-level programming languages, deployed over individual containers, predominantly adopting the database per microservice pattern and asynchronous event-based communication. By investigating several microservice deployments, we reveal that data management is key to achieving microservices' desirable properties, such as achieving scalability through functional decomposition and data partitioning, increasing data availability

via fault isolation and data centralization, faster software evolution via independent schema management, and event-driven data management as opposed to classic pull-based data querying.

Although microservices are supposedly independent, they often end up depending on each others' data, leading developers to encode cross-microservice coordination mechanisms that lead to data corruption, to resort to weaker guarantees (i.e., eventual consistency) when replicating data across microservices, to eschew data integrity constraints cutting across microservices, and to fall prey to the lack of isolation guarantees when joining data across microservices. By pursuing a model of decentralized data management, practitioners end up encountering challenges that should have been solved by database systems.

We specify a set of characteristics and features for futuristic database systems to embrace the needs of microservice applications by design. Directions for devising a solution include but are not limited to supporting distributed transactions while allowing microservices to remain loosely coupled, ensuring correctness for constraints cutting across microservices (e.g., referential integrity) and event processing, and consistently querying and replicating microservices' states without breaking encapsulation.

However, these must account for the programming abstractions that practitioners are used to working with, necessitating a cross-domain approach, combining databases, distributed systems, and software engineering perspectives to realize an effective and efficient microservice-oriented database system. As a result, this exploration shed light on the limitations of state-of-the-art data management systems and paved the way for the next work streams.

This contribution is found in Chapter 3.

Contribution #2. Inspired by the uncovered data management requirements and challenges of our first contribution, we found that microservice benchmarks target problems that are either oblivious to data management or position data management as not a primary concern, thus failing to reflect the complexities of data management in real-world microservices.

They do not prescribe distributed transactions, data replication, data and event querying and processing, and data integrity constraints, which are predominant data management requirements in microservice architectures. Consequently, this gap makes it difficult to effectively advance data system technologies to support microservice applications.

To bridge this gap, we propose *Online Marketplace*, a novel data manage-

ment benchmark modeling a popular microservice application scenario containing eight microservice types, ten message types, four complex transaction types, three query types, reflecting key data management tasks pursued by practitioners, such as distributed transaction processing, data replication, consistent queries, event processing, and data constraint enforcement.

To reflect the data management challenges encountered in practice, *Online Marketplace* prescribes several data management criteria that a data management platform should meet, including functional decomposition, resource isolation, transactional isolation levels, data replication semantics, event processing correctness, query processing consistency, and data invariants of microservices. These criteria allow for a fair comparison between different systems on the same basis while meeting the complex nature of deployments in industry settings. Additionally, *Online Marketplace* features a driver that addresses the challenges of creating workloads that accurately reflect the dynamic and distributed state of the microservices.

Case studies demonstrate the effectiveness of *Online Marketplace* in uncovering performance issues and pinpointing the missing core data management features in state-of-the-art data platforms. Additionally, it provides a concrete example for database researchers to understand the data management challenges to work on, serving as a testbed for experimenting with novel algorithms, approaches, techniques, programming models, etc. To our knowledge, *Online Marketplace* is the first microservice benchmark that embraces core data management requirements sought by microservice practitioners. Notwithstanding the focus on microservice architectures, the case studies demonstrate that the benchmark also serves as a test bench for other trending cloud architectural styles and programming models, including modular monolithic architectures, Function-as-a-Service (FaaS), and actors.

This contribution is found in Chapter 4.

Contribution #3. Different competing runtimes for distributed applications in the cloud emerged recently, many claiming to support microservice applications. Implementation issues encountered while implementing the *Online Marketplace* benchmark requirements suggested a mismatch between state-of-the-art data platforms and the practice, which could only be fulfilled through weaving together heterogeneous systems, confirming our earlier findings (Contribution #1).

Given the resemblance of programming paradigms and data models but substantial differences in system design, terminologies, and consistency semantics, we found it opportunistic to review and discuss state-of-the-art

cloud runtimes concerning data management and modern software engineering requirements.

We propose a taxonomy that embraces programming models, state management, and application lifecycle, reflecting the building blocks practitioners use while implementing applications in the cloud. Through the taxonomy, we characterize the highly unstructured and heterogeneous cloud application landscape, exploring the different designs and limitations. Overall, this contribution synthesizes and organizes the cloud programming landscape for transactional applications, fostering the advancement of database research.

This contribution is found in Chapter 5.

Contribution #4. Investigating the nature of data management tasks and how data management logic is encoded in microservice applications revealed a surprising aspect.

Microservices operate primarily outside the database system boundary, with a substantial number of data management tasks at the application layer and communicating and exchanging data through application-generated events. The underlying microservice databases are oblivious to the complex interplay of microservices and data flowing outside the database and hence are fated to play a secondary role, mostly relegated to only providing data durability.

The traditional application programming interfaces offered by database systems treat applications as black boxes [271]. Therefore, it is necessary to open this black box to understand the complex interplay among microservices outside the database system, effectively support data management tasks natively, and enforce application safety, among the most challenging tasks microservice developers face.

We argue that appropriate abstractions allow for enriching the knowledge about the application's data management logic and the complex interplay of microservices. We propose the virtual microservice, a database abstraction that reflects the inner characteristics of a microservice running outside the database, including inbound and outbound events, private state, foreign data dependencies, and integrity constraints. As the internal metadata and semantics of microservices are made available, the database system gains knowledge about the event flow across microservices and the constraints cutting across microservices. The application is no longer a black box to the database.

We demonstrate how virtual microservices are seamlessly integrated into the application layer through familiar programming constructs without break-

ing the desired encapsulation principle and how data management tasks are transparently identified for being pushed downward to the database. The approach can effectively prevent ad hoc synchronization mechanisms at the application level, alleviating the burden on developers and favoring a proper division of tasks between the application layer and the database system.

Contribution #5. The benefits brought about by virtualized microservices cannot be met without native database-level support. In this contribution, we demonstrate how to enable pushing data management tasks to the database and materialize the virtual microservice abstraction into a principled database architecture. This must, however, account for the tiered design of modern applications while meeting key principles of microservices by providing isolation boundaries between virtual microservices and decentralized data management.

With a proper data management benchmark and a proper database abstraction, we close the database system gap (**#4**) by developing virtual Microservice Oriented DataBase (vMODB), a novel distributed DBMS that tackles the problem of data inconsistency in distributed and asynchronous applications by design.

We propose a novel cross-stack architecture, encompassing an application framework, which controls the interaction of a microservice with the system, and specific-purpose components that target particular data management concerns and scale independently. An application framework is a key component. It identifies the application semantics and data management logic, and informs the database about the data management tasks carried out by the application. Most importantly, it takes control of the application [88], a necessary condition to control the scheduling order of functions and enforce transactional isolation and application constraints. In other words, the database now transcends the database tier scope to offer application safety benefits to microservice programmers.

The state of practice today is characterized by practitioners gluing together several heterogeneous components and message brokers in an ad-hoc manner, giving up transactional guarantees and creating data consistency issues. vMODB removes this burden by unifying event logs and application state management into a common event-driven execution framework and enforcing ACID semantics transparently from user implementations while maintaining the benefit of stateless program design pursued by developers. Experiments show vMODB outperforms 3x state-of-the-art distributed application frameworks while providing key consistency semantics such as seri-

alizable transactions and snapshot isolation to ad-hoc queries.

Chapter 6 envisions a novel type of database system providing a virtual microservice programming interface and Chapter 7 presents a concrete formalization and implementation details of a virtual microservice oriented database system.

1.3 Publications

This thesis is based on the following manuscripts:

1. Chapter 3 is based on a published conference paper [150]: Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. "Data management in microservices: state of the practice, challenges, and research directions". In: *Proc. VLDB Endow.* 14.13 (Sept. 2021), pp. 3348–3361. ISSN: 2150-8097. DOI: 10.14778/3484224.3484232. URL: https://doi.org/10.14778/3484224.3484232

2. Chapter 4 is based on a published conference paper [146] and a talk in a conference [147]:

   Rodrigo Laigner, Zhexiang Zhang, Yijian Liu, Leonardo Freitas Gomes, and Yongluan Zhou. "Online Marketplace: A Benchmark for Data Management in Microservices". In: *Proc. ACM Manag. Data* 3.1 (Feb. 2025), pp. 1–26. DOI: 10.1145/3709653. URL: https://doi.org/10.1145/3709653

   Rodrigo Laigner and Yongluan Zhou. "Benchmarking Data Management Systems for Microservices". In: *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 2024, pp. 5671–5672. DOI: 10.1109/ICDE60146.2024.00467. URL: https://doi.org/10.1109/ICDE60146.2024.00467

3. Chapter 5 is based on a tutorial presented as part of a conference [142]: Rodrigo Laigner, George Christodoulou, Kyriakos Psarakis, Asterios Katsifodimos, and Yongluan Zhou. "Transactional Cloud Applications: Status Quo, Challenges, and Opportunities". In: *Proceedings of the 2025 International Conference on Management of Data*. SIGMOD '25. Berlim, Germany: Association for Computing Machinery, 2025

4. Chapter 6 is based on a published conference paper: Rodrigo Laigner, Yongluan Zhou, and Marcos Antonio Vaz Salles. "A Distributed Database System for Event-Based Microservices". In: *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*. DEBS '21. Virtual Event, Italy: Association for Computing Machinery, 2021, pp. 25–30. ISBN: 9781450385558. DOI: 10.1145/3465480.3466919. URL: https://doi.org/10.1145/3465480.3466919

5. Chapter 7 is based on a manuscript [148]: Rodrigo Laigner and Yongluan Zhou. *vMODB: Unifying event and data management for distributed asynchronous applications*. Manuscript under review. 2025. URL: https://arxiv.org/abs/2504.19757

The results of this thesis have also inspired other works, leading to published journal and conference papers [141, 156, 151, 63] that are addressed along Chapter 8.

1.4 Structure of Dissertation

This dissertation is organized as follows. Chapter 2 provides the necessary background. Chapters 3, 4, 5, 6, and 7 present the five contributions summarized above, respectively. It is worth noting that the state of the art is presented and discussed for each contribution separately. Chapter 8 concludes with a reflection about this dissertation and outlines ongoing and future work.

Chapter 2

Background

2.1 Transactional Database Management Systems

2.1.1 Transaction Processing

A transaction in a database management system (DBMS) is a logical unit of work within a program, represented as a series of actions that include reads and writes of database objects, delimited by a begin and commit/abort action [99]. DBMSs typically interleave the execution of operations across multiple transactions for performance reasons. To prevent users from reasoning about all possible interleavings of concurrent transactions, their potential impact on program correctness, and failure scenarios, a typical guarantee provided by transactional DBMSs is consistency. When referring to consistency, literature and DBMS vendors often mean ACID transactional consistency [99, 18], an acronym for the following properties:

- Atomicity: all or nothing guarantee, meaning either all actions of a transaction succeed or none. If a transaction aborts or experiences a system failure, all the effects of its actions are undone;
- Consistency: guarantee that preserves the consistency of the database, i.e., a transaction can only bring the database from one consistent state to another;
- Isolation: it guarantees that concurrent transaction executions appear equivalent to some serial order;

- Durability: it guarantees that the effects of a transaction, once committed, are persisted even if the system crashes.

It is worth noting that the isolation property of the ACID acronym is often assumed to be serializability [99]. However, many systems default to weaker isolation levels such as Snapshot Isolation or Read Committed [17]. In sum, while a transaction running under serializable isolation may still be exposed to the effects of concurrent transactions that logically precede it in the serial order (even if they overlap), users can reason about correctness as if transactions ran serially. As a result, programs are exempted from implementing concurrency control over database objects.

2.1.2 Concurrency Control

DBMSs typically achieve serializability via a concurrency control protocol, which is classified into two categories, pessimistic and optimistic [99].

Pessimistic protocols avoid conflicts on data access via locking database objects. The most widely adopted pessimistic protocol is 2-Phase Locking (2PL), which acquires locks in a growing phase and releases them in a shrinking phase, during which no further locks are acquired.

In 2PL, locks are acquired either during transaction execution or conservatively, where all database objects accessed are locked at the beginning of the transaction. While the conservative mode prevents deadlocks during transaction execution, disclosing all database objects accessed before a transaction begins is often challenging, since some data accesses are only revealed through complex program logic or indirectly, e.g., via secondary indexes.

The shrinking phase can be either strict or not. When strict, locks are released only at the end of transaction execution; when non-strict, locks are released as soon as the transaction signals that no new locks will be acquired (e.g., at commit time, when transaction actions have not yet made durable changes to the database state). While non-strict shrinking often yields higher performance by allowing concurrent transactions to access previously acquired locks faster, transaction aborts can trigger cascading rollbacks, affecting a whole chain of transactions that acquired locks released during the non-strict shrinking phase. As a result, the non-conservative and strict 2PL, often referred to as S2PL, is the most widely adopted variant.

In optimistic concurrency control, the DBMS does not resort to locks to control concurrent access to database objects during transaction execu-

tion [139]. The core assumption is that in scenarios with little or no contention, the overhead and blocking from managing locks outweigh their benefits. In this case, by tracking the database objects accessed during the transaction, the transaction's read-and-write set can be validated against overlapping transactions upon completion. If conflicts are detected, one of the conflicting transactions is aborted [139, 99]. While optimistic concurrency control can perform well for low-contention, read-dominated workloads like online analytical processing (OLAP), its performance degrades under heavy transactional scenarios with high contention or long transactions due to the overhead of validating and aborting a substantial number of transactions [99].

2.1.2.1 Distributed Transactions

While ACID properties provide a sufficient condition for ensuring database correctness for a single process (or single-site DBMS), ensuring all-or-nothing atomicity for a transaction that performs updates in multiple processes requires a coordination protocol [115].

The Two-Phase Commit (2PC) protocol is the most widely adopted atomic commitment protocol for distributed transactions. It is divided into two phases: the prepare and commit/abort. A transaction processing agent or a participant process, typically the one that initiates or ends the transaction execution, assumes the role of coordinator.

The coordinator first sends a `prepare` request to all other participant processes upon the transaction's completion. After each participant processes replies with a `commit` or `abort` message, the coordinator decides the outcome. Only if all processes vote to commit is the distributed transaction considered safe to commit. When voting on a commit, the participant process logs the decision to ensure the vote survives a crash. On the other hand, if at least one participant process fails to reply within the time window or replies with an abort message, the coordinator aborts the transaction. Next, the coordinator logs the outcome and initiates the second round of communication by sending the `outcome` message to all participant processes, who then log the decision, release locks, and acknowledge that the outcome has been processed.

Criticisms of the 2PC include network round-trips, storage I/O costs at both coordinator and participants, and the protocol's inherent blocking under failures. In particular, if the coordinator fails during the protocol execution, prepared participants are unsure of the transaction's outcome. Typical protocol optimizations include batching log flushes [99], running the

protocol in epochs to amortize the overhead of committing each transaction in isolation (i.e., group commit) [160], and early lock release [73].

2.1.3 Deterministic Transaction Processing

At scale, coordination protocols tend to exhibit high latencies [115]. The combined effects of S2PL and 2PC on controlling access to database objects and committing transactions, respectively, add overhead to the critical path of distributed applications.

> The first principle of successful scalability is to batter the consistency mechanisms down to a minimum, move them off the critical path, hide them in a rarely visited corner of the system, and then make it as hard as possible for application developers to get permission to use them.[110]

Deterministic transaction processing is characterized by defining a global ordering of transactions as input and having an execution model that prescribes an execution equivalent to that specific order, possibly exploiting high concurrency [250]. For that, deterministic transaction processing in DBMSs requires that transactions disclose their read and write set before entering the system (similarly to conservative 2PL) and being assigned a transaction identifier (i.e., recorded to disk), and that the transaction logic is deterministic (i.e., it runs to completion once it enters the system).

Taken together, these two assumptions provide a powerful framework for reducing coordination overhead and simplifying atomic commitment. By reasoning about the read-and-write set of transaction inputs, the system can predefine a lock-granting order that prevents deadlocks from forming, thereby ruling out aborts caused by conflicts. By executing program logic deterministically, a failure does not affect the outcome of a transaction execution, since a failed process can simply restart from the last committed transaction, simplifying the atomic commitment protocol to a single network round-trip.

However, in many cases, the first assumption is not possible, such as when a secondary index lookup is required to determine which key to access [250]. This limitation requires a *reconnaissance* phase, in which the transaction runs against a database snapshot and the read and write set is collected [251]. Upon collection, the transaction is executed and committed

only if the read and write sets from the *reconnaissance* phase and the actual execution show no divergences. In case of non-deterministic program statements (e.g., obtaining a random number or current time milliseconds), the system must resolve the values and include them as part of the transaction arguments [250]. Albeit leading to substantial performance improvements over traditional distributed transaction protocols [251], deterministic databases have not arisen as popular solutions in industrial settings.

In the meantime, deterministic transaction execution has shown a good fit to actor systems [157, 156]. The reason is that, as data are partitioned in relation to the actors they belong to, access to actors naturally maps to access to data. However, similarly to cases where the read and write sets are not known in advance, the actual actors who access may not be known in advance.

This thesis explores deterministic transaction execution in event-driven microservice architectures 2.2.3 under less restrictive assumptions, reasoning about high-level application components rather than data accesses, thus refraining users from disclosing the read-and-write sets of their transactions, as discussed in Chapter 7.

2.1.4 Implementing Transactional Programs

2.1.4.1 Stored Procedures

Stored procedures are database programs that allow users to express application logic following an imperative paradigm, with complex control structures such as loops and conditional statements [99]. Once compiled, it can reuse parsing and planning for the set of SQL statements specified in the program body across disparate executions and make more effective use of database resources, leading to performance benefits compared to executing interactive transactions (submitting each SQL statement separately) [122].

2.1.4.2 Transactions in Modern Applications

The dominant design in applications that process transactions and serve queries to many users at low latency typically relies on relational databases, abstracted through a database connectivity interface.

The reasoning is twofold. A relational database provides complex querying features, indexing, and key safety properties such as atomicity and trans-

action isolation, which prevent developers from dealing with failure handling and concurrency in the application code [99]. Meanwhile, developers prioritize encoding complex application logic separately from the database to leverage advanced programming language features, better isolate resources, and promote loose coupling, enabling faster maintenance and evolution of application features [249].

To better map application objects and query declarations to relational database connectivity interfaces, applications often rely on object-relational mapping (ORM) frameworks such as Hibernate [119]. ORM frameworks offer metaprogramming features, such as annotations, to allow developers to express relational constraints (e.g., primary and foreign keys) through application objects and transactional properties, such as whether operations within a block of code (e.g., a method) must be executed as a read-only or read-write transaction.

Database connectivity interfaces, such as those based on standardized protocols like JDBC [187], are provided so applications can connect to and interact with the DBMS. Consequently, apart from reads and writes submitted by the application, the DBMS is oblivious to application semantics that execute in the application level.

2.1.4.3 Application-defined Transactions vs Stored Procedures

Application components (i.e., application logic) can be ported or implemented as stored procedures to enable remote invocation. However, this has not emerged as a popular choice for many reasons [26, 199].

A growing number of applications are modeled via and operate over objects, a natural model for entities in the multi-player, Internet of Things (IoT), e-commerce, social, and mobile application domains [25]. These objects are long-lived and distributed across servers, communicating via message-passing semantics, reflecting their real-world interactions [26].

Encoding business logic in stored procedures is also challenging for developers because it breaks applications' tiered architecture, making it difficult to test, verify, troubleshoot, and maintain application artifacts. Furthermore, having all components and associated software artifacts managed and executed within the DBMS can hinder achieving key principles of event-driven and microservice architectures, such as loose coupling, fault isolation, and independent deployment of components [179].

2.2 Distributed Application Architectures

Distributed application architectures are the cornerstone of large-scale applications. In this section, we discuss historical developments and key paradigm shifts observed over the last few years.

2.2.1 N-tier Architectures

N-tier Architectures promote a model in which clients, compute, and storage are disaggregated to reap the benefits of loose coupling, failure isolation, and independent life cycles.

Initially, it was common to have application logic encoded as stored procedures and data stored in a database server serving multiple clients, characterizing the so-called two-tier architecture. This setup was particularly driven by the lack of reliable networks, which motivated the co-location of application functionality and data on the same machine [29].

Apart from its apparent simplicity, the two-tier architecture can pressure the computational resources of the database server in the presence of an increasing number of client requests and growing data [189]. In particular, each client requires its own database session. As sessions are linked to a database connection, an often expensive resource, and there is no possibility of multiplexing multiple clients in existing connections, computational resources can be quickly drained. Scaling up the database server emerged as a popular performance strategy. However, this would often involve overprovisioning resources and increased costs [36]. Scaling out the database has not emerged as an attractive strategy at the time because it would exacerbate the challenges with unreliable networks, not to mention burden clients with database node availability concerns.

The advent of TCP/IP and the advance of application server technologies motivated the emergence of the three-tier architecture. By promoting a model where clients, application functionality, and data are decomposed into presentation (aka client), middle (aka application), and database tiers, respectively, each tier can fail, scale, and provision resources independently, according to their particular evolving requirements [125] The idea is that different tiers share responsibility for the application's availability, scalability, and performance, allowing specific-purpose optimizations in each tier. The middle tier plays a key role. By representing the application, it sits between

the presentation, which dispatches client requests, and the database tier, which stores the application state.

The fast adoption of the three-tier architecture over the last decades has also been credited to a better adherence to modern development practices [26], including but not limited to object-oriented programming and the use of application runtimes (e.g., Java Virtual Machine and .NET) that prevent developers from managing challenging application concerns such as memory allocation and deallocation, object life-cycle management, thread scheduling, portability, etc. It is also reported that key software engineering tasks, such as change management, code evolution, feature reuse, and software maintenance, are facilitated by managing application artifacts (e.g., source code, settings, tests) independently of the database tier [26].

Typically, in this setup, application functionalities are encoded in the middle tier and no longer as stored procedures [249, 199, 18]. Whenever data requires being updated, it is pulled from a DBMS, modified, and submitted to the DBMS as an update operation, being always transient in application memory. In other words, the responsibility for application state durability is entirely offloaded to an external database system. This setup characterizes the stateless middle-tier design [27].

2.2.2 Cloud Computing and Distributed Applications

Cloud computing is a model for renting computational resources. Resources (e.g., compute and storage) and services (e.g., message queues and databases) can be acquired on a pay-per-use basis, providing higher flexibility and cost savings for users and organizations.

Cloud providers are responsible for acquiring and maintaining data centers, servers, and services, whereas software teams, freed from resource management, are responsible for provisioning the right amount of resources and making applications operational.

Initial cloud computing offerings were based on renting physical and virtual machines (VMs), often involving over-provisioning, that is, provisioning more resources than necessary to possibly accommodate dynamic workloads. Additionally, for every new machine provisioned, operating system (OS) configurations and application deployment procedures must be reproduced and adapted to different computer architectures. Application maintainers are also responsible for detecting failures, restarting, and applying recovery routines, ensuring applications remain operational upon crashes.

To alleviate these challenges and better support the transition to cloud, specialized cluster management platforms emerged, such as container orchestrators like Kubernetes (also known as K8s) and Docker Swarm. These orchestrator platforms abstract away physical resources of a cluster (e.g., CPUs, memory, and storage) through logical resources called containers.

A container is a lightweight execution unit that often represents an application process, abstracting away the underlying computing environment, thus allowing for faster and more reliable software deployments across heterogeneous environments. Containers share the underlying machine's OS system resources and, unlike VMs, do not require an OS per application. A container is configured using a container image, an executable unit of software that packages an application's codebase and all dependencies, along with the runtime (e.g., Java Virtual Machine), system tools, system libraries, and settings. Container images are executed by container runtimes (e.g., Docker Engine) and then become managed containers on an orchestrator platform.

Users can configure auto-scaling and self-healing policies for containers, making it easier to meet availability metrics for applications such as uptime. Furthermore, orchestrator platforms can typically migrate containers across managed machines to balance load and prevent specific machines in the cluster from becoming overloaded. Besides, upon a container failure, they make a best effort to allocate the failed container to an available resource. The autonomy of each container also aid to software maintenance. Containers can be deprecated, upgraded, and deployed independently without affecting executing containers. As a result, container orchestration is of key importance in modern distributed application development in the cloud.

2.2.3 Event-driven Microservice Architectures

The cloud computing paradigm has impelled industry practitioners to rethink how applications are designed and deployed. In particular, to bridge the aspiration for on-demand, fine-grained resource provisioning enabled by the cloud and modern software engineering practices, microservice architectures have emerged as the most popular alternative in the cloud application development landscape [235].

Opposing the conventional monolithic architecture encountered in the three-tier architecture, where application modules are packaged as a single executing unit and interact via function calls, a microservice architecture promotes designing modules as independent components that interact with

each other via communication mechanisms such as HTTP-based protocols or asynchronous events [179].

Microservices can use different programming languages and independent databases through a decoupled design, leading to the so-called decentralized data management principle. Besides, microservices promote fault isolation, in which failures in individual microservices are not propagated externally, e.g., in a container-based deployment. However, partitioning an application's state into distinct, independent components presents challenges. Although supposedly designed to operate autonomously, microservices often exhibit data and functional dependencies among themselves [150]. To manage such dependencies while maximizing decoupling, practitioners often rely on asynchronous, event-based communication between microservices rather than synchronous ones, such as remote procedure calls [143, 133].

An event serves the purpose of exchanging data, triggering remote computations, and communicating failures [150, 196]. In this case, persistent asynchronous communication is preferred since it does not require microservices to remain active during message transmission [248]. Communications remain loosely coupled in time, promoting a model in which microservice states eventually converge, following a design resembling the BASE approach [204].

Despite the apparent simplicity, in practice, functionally partitioning applications and eschewing ACID properties (§ 2.1.1) creates challenges in ensuring data integrity. First, as there is no longer a single database schema, this often implies moving data integrity constraints out of the database and into the application [204]. Second, developers must make their application logic idempotent to mitigate the effects of partial failures, such as redelivery and reprocessing of messages. Third, as messages are processed at the application level, it is the responsibility of developers to ensure the proper isolation across dequeuing and queuing tasks so to ensure ordering semantics.

2.3 Distributed Application Frameworks

Containers facilitate the setup, execution, and maintenance of distributed applications by deploying services seamlessly across disparate computer architectures, load balancing and failure handling through container migrations across nodes, and applying auto-scaling and self-healing measures. However, because microservices and BASE are sets of principles for designing partitioned applications rather than full-fledged data management solutions,

many operational challenges remain open. Application maintainers must still manage the lifecycle of application objects, implement recovery routines, ensure message delivery and processing guarantees, and maintain application state consistency, often by weaving together many heterogeneous systems [114].

These challenges motivated the emergence of distributed application frameworks. Popular web frameworks such as Spring [233] incorporated libraries that enable message exchange via message-oriented middleware, such as Kafka [231]. These extend the framework with programming constructs to facilitate event production and consumption and to acknowledge processed events. However, these extensions do not relieve developers of the responsibility to ensure event processing correctness, such as achieving exactly-once processing guarantees.

Dapr [171] also emerged as a prominent framework in this realm, offering developer productivity facilities for observability and deployment, as well as generic APIs allowing the use of different messaging systems and programming languages. Dapr features transparent event consumption tracking for applications, using automatic retries to achieve at-least-once delivery guarantees. However, it cannot guarantee exactly-once processing, which may indefinitely leave non-idempotent applications in an inconsistent state. Furthermore, Dapr offers a centralized, key-value state management abstraction but does not support inter-component ACID transactions. As a result, developers are still responsible for ensuring application state consistency, and no duplicate events are processed in the event of failures. Other frameworks offer stronger execution guarantees [104, 152], however, as Dapr, they do not offer ACID.

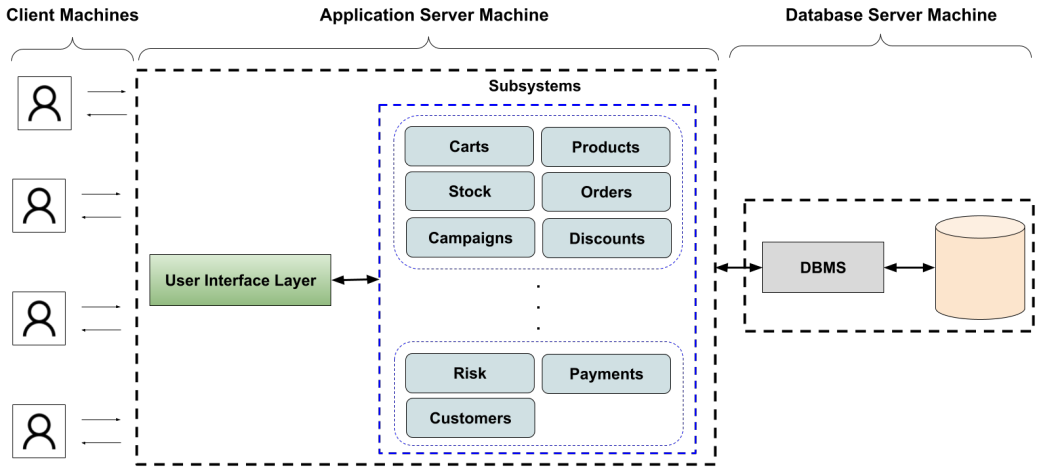
Actor and function-as-a-service (FaaS) systems are also often considered frameworks for facilitating the development of event-driven and microservice applications. Unlike Dapr, they enforce a single-threaded programming model, requiring partitioning the application state in a particular way [34, 31], which can be restrictive for a broad range of data-intensive applications and developers [190], not to mention the lack of indexing, query processing, and relational data model. Recent work has found that Actor and FaaS systems exhibit performance limitations in complex transactional applications [157, 156, 146]. Chapter 5 expands this discussion with a characterization of these frameworks.

Chapter 3

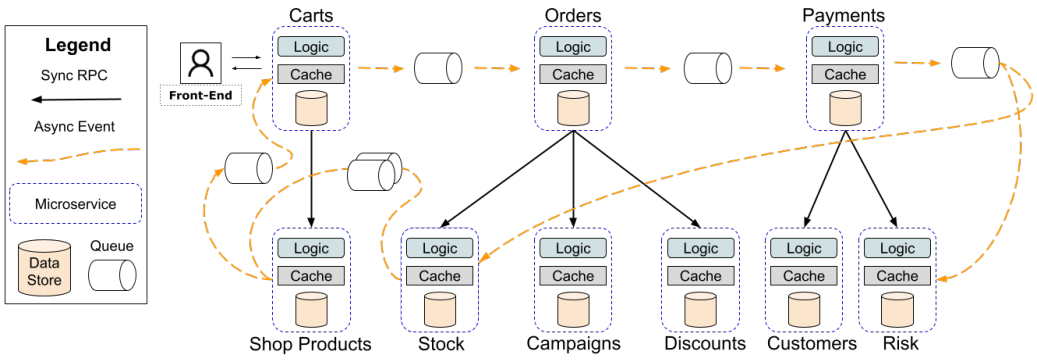
Data Management in Microservices: State of the Practice, Challenges, and Research Directions

Microservices have become a popular architectural style for data-driven applications, given their ability to functionally decompose an application into small and autonomous services to achieve scalability, strong isolation, and specialization of database systems to the workloads and data formats of each service. Despite the accelerating industrial adoption of this architectural style, an investigation of the state of the practice and challenges practitioners face regarding data management in microservices is lacking. To bridge this gap, we conducted a systematic literature review of representative articles reporting the adoption of microservices, we analyzed a set of popular open-source microservice applications, and we conducted an online survey to cross-validate the findings of the previous steps with the perceptions and experiences of over 120 experienced practitioners and researchers.

Through this process, we were able to categorize the state of practice of data management in microservices and observe several foundational challenges that cannot be solved by software engineering practices alone, but rather require system-level support to alleviate the burden imposed on practitioners. We discuss the shortcomings of state-of-the-art database systems regarding microservices and we conclude by devising a set of features for microservice-oriented database systems.



(a) Traditional monolithic architecture



(b) Microservice architecture

Figure 3.1: Monolithic vs. microservice architectures

3.1 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 [89, 179]. In particular, we are witnessing the increasing adoption of microservice architectures to replace the traditional monolithic architecture (Figure 3.1). In contrast to a monolithic architecture (Figure 3.1(a)), where modules and/or subsystems are integrated and cooperate in a centralized manner, a microservice archi-

tecture, (Figure 3.1(b)), organizes an application as a set of small services that are built, deployed, and scaled independently.

In the scenario of Figure 3.1, we depict an e-commerce application to contrast both designs. In the monolithic case, to process an order, for example, the *Cart* module performs a function call to the *Order* module, which then performs additional function calls to the *Stock*, *Campaigns*, and *Discounts* modules to safeguard the order is correctly placed. In contrast to direct function calls between modules, microservices communicate with each other through remote calls, such as HTTP-based protocols [89, 97] or asynchronous messages [288]. 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 [288].

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 [90]. For instance, the *Products* database may take advantage of a document-oriented model to allow for agile schema evolution, whereas the *Stock* microservice may rely on the relational model to safeguard constraints over stock items. As a result, 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 into small parts.

Motivation. Despite the increased adoption of microservices in industry settings [57, 138, 111, 105, 216, 163, 167, 14, 158] and the perception that data management is a major challenge in microservices [134, 261, 94, 162, 278, 227], 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 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.

Methodology. To bridge this gap, this paper presents an investigation of the state of the practice of data management in microservices. 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. From 300 peer-reviewed articles analyzed, 10 representative articles [167, 163, 14, 216, 111, 138, 57, 48, 260, 143] were selected for review; (ii) we analyze 9 popular microservice-based applications [215, 220, 243, 76, 284, 267, 232, 23, 266], selected out of more than 20 open-source projects, and; (iii) we design an online survey to gather the opinions of developers and researchers experienced with microservices in real-world settings, allowing us to cross-validate the findings of the previous steps. In total, more than 120 practitioners provided important information about their microservices' deployments in industry settings. Taken together, these three interrelated explorations provide new and comprehensive evidence on data management practices and challenges in microservices. Additional details about our methodology can be found in [145].

Findings. From our investigation, we observed 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 functionality and private state dependencies amongst each other.

As a result, developers have a hard time reasoning about enforcement of application safety within and across microservices. The latter relates 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 (DBMSs) and end up weaving together several heterogeneous data systems such as message brokers, in-memory caches, analytical engines, and loosely structured and structured DBMSs in an ad-hoc manner. This system complexity leads to a substantial amount of data management logic at the application layer to meet the data management requirements of microservices.

As a consequence, database systems no longer play a central role in this novel paradigm, often relegated to providing data storage functionality. The lack of a holistic view, by means of an unawareness of the dataflow, constraints, and the complex interplay among microservices, leads to the impos-

sibility of effectively ensuring data and application safety in the microservice paradigm.

Contributions. In summary, we make the following contributions:
(i) We survey a gamut of experienced practitioners, the literature, and popular open-source microservice applications to characterize the state of the practice of data management in microservices. The findings (presented along § 3.3, § 3.4, § 3.5, and § 3.6) reveal several ad-hoc data management practices in microservice architectures never reported before in the literature, suggesting that developers are insufficiently served by state-of-the-art DBMSs.
(ii) We illustrate through source code snippets, extracted from popular open-source microservices, challenges that practitioners face when trying to meet the data management requirements of microservices (§ 3.5). These challenges are also highlighted and extended by experienced microservice developers when asked about their most pressing pains regarding data management (§ 3.6), thus illustrating issues that database technology should aim to address.
(iii) Based on the observed data management practice and challenges, we present a set of features for future database systems to tackle the data management requirements of microservices by design, so they can play a central role in this new paradigm (§ 3.7). We discuss why the state of the art is not able to address all the challenges in conjunction and we conclude by suggesting how the features can be realized into a microservice-oriented DBMS.

The three contributions together provide new directions for the database community to start leading the efforts on data management for microservices. Our goal is to inform engineers of future data management systems about the unmet needs of an emerging application paradigm by providing comprehensive evidence of shortcomings and pitfalls microservice developers face when dealing with data management. We view this work as an example of engaging with database system users to understand and characterize the practice in order to derive new research opportunities for emerging applications. We hope the results drive the reflection of our community towards effectively meeting the needs of microservice-based applications.

3.2 Related Work

Despite the extensive work in microservice architectures in other communities, an in-depth analysis of data management in this context is lacking in

existing research. Works in software engineering focus on migrating from monolithic architecture to microservices [39, 38] or investigating other general software engineering aspects [101, 261, 94, 278, 134], 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 the data management community builds, fostering further research.

In addition, our work provides an in-depth characterization of challenges faced by microservice developers while implementing data management logic in the application-tier. In regard to this contribution, although some studies described related pitfalls, such as *shared persistence* [246, 200], and previous literature investigated architectural smells and anti-patterns in microservices [177, 38, 247, 252], 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 paper.

Although there are initial works tackling specific challenges of microservices [63, 151], they fall short providing a holistic solution. In light of the many unveiled data management challenges in microservice architectures, our work is the first to reflect on core limitations that prevent database systems from adequately serving the needs of this emerging architectural style.

3.3 State of the Practice

In this section, we focus on characterizing the motivating factors for data management in microservices as well as the most prominent DBMSs and deployment patterns employed.

3.3.1 Motivating factors

Existing works [89, 97, 39, 111, 179] argue 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 – particularly by means of use of a database/schema per microservice. Thus, these motivations are intrinsically

Table 3.1: Motivations for microservices in data management

Motivation	#	%
Scalability through functional decomposition	55	35.71
Fault-isolation (e.g., increasing data availability)	32	20.77
Agility on data change (e.g., facilitating schema evolution)	32	20.77
To enable event-driven data management (e.g., as opposed to classic pull-based data querying)	23	14.93
Polyglot persistence	9	5.84
Others	3	1.94

related to data management, and decentralized data management is a major foundation in the adoption of microservices.

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). The provided options were influenced by the following: (i) a preliminary assessment with four industry representatives with strong background in microservices through interviews; (ii) the literature review; (iii) the analysis of open-source repositories; and (iv) discussions among the authors. Table 3.1 shows the options provided and respective responses. We highlight the following important observations (referenced hereafter by **O#**).

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 [167, 163, 216, 111, 48, 57, 143]. 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 [204], a strategy that involves "grouping data by function and spreading functional groups across [different] databases."

O1. The 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.

Decentralized data management: Practitioners developing applications in real-world settings largely deal with evolving requirements and subsequent schema changes [260]. 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 [163, 111, 138, 48, 260]. Surprisingly, although polyglot persistence derives naturally from decentralized data management, it is the least cited motivation in both literature [14, 138, 260] and the survey, being roughly 3.5 times less cited by practitioners than schema evolution.

O2. The results suggest that the support for a large variety of data models is less of a pressing need than schema evolution in microservices.

Event-driven microservices: Event-driven systems constitute an emerging trend in the design of data-driven software applications [133]. Microservices are often mentioned as a compelling paradigm for designing event-driven applications [131, 222, 143]. Almost 15% of the participants consider event-driven data management a primary motivation for microservices, which may indicate a trend in industry adoption. Most papers [167, 163, 216, 111, 138, 48, 260, 143] mention the use of asynchronous primitives for message-oriented communication to achieve loose coupling among microservices and facilitate decentralized data management. The same trend is encountered in open-source repositories [284, 220, 23, 215, 76, 215, 243].

O3. The results suggest that asynchronous events should emerge as a concern when devising data management technologies for microservices.

Summary. Our results indicate that functional decomposition, fault isolation, schema evolution, and event-driven architecture are the primary reasons behind the adoption of microservices for data management.

3.3.2 Database systems 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 [169]. We asked the participants which database patterns they use 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.

We observe that most practitioners prefer encapsulating a microservice's 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, 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 repository 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.

O4. The results suggest that microservices are prevalently deployed in individual containers, predominantly using the database-per-microservice pattern to achieve performance and fault isolation.

Furthermore, 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 adoption of multiple DBMS belonging to at least two of the provided categories is often reported. For instance, 47.97% indicated the use of at least one relational DBMS in conjunction with MongoDB. Besides, we observed a trend (15%) of use 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 DBMSs for the underlying microservice databases, Redis as a caching layer to provide fast data access to recurring requests, and replication of data through an event-driven approach to ElasticSearch for fast online analytical queries. Overall, the results of DBMS adoption from the survey are very aligned with the reviewed papers and the open-source repositories.

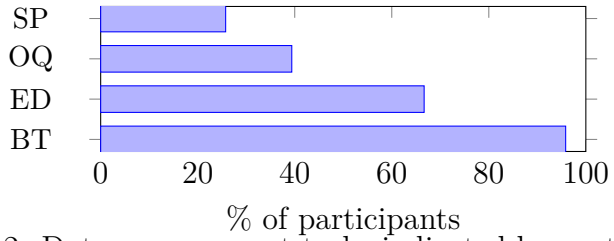


Figure 3.2: Data management tasks indicated by participants

3.4 Data Management Logic In Practice

Continuing from our last section, we were also interested in characterizing the types of queries and data management logic that are performed in microservice-based applications. 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 how business logic is implemented in their applications. Figure 3.2 shows the types of data management logic identified from the responses, characterized by Cross-Microservice Business Transactions (BT), Event-Driven Computing (ED), Online Queries (OQ), and Stream Processing (SP).

3.4.1 Cross-microservice business transactions

The studies reviewed from literature [167, 260, 216, 111, 57], the open-source repositories [76, 23, 243, 284, 267, 232, 266, 215], 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 [43, 18]. Similar to Pavlo's findings [199], we did not find evidence of the use of stored procedures or the configuration of databases in serializable isolation. The same applies to literature.

Regarding business transactions across microservices, [1] conversations (i.e., the interaction between a set of consumer and producer services) are prevalent. Hohpe [120] argues that orchestration and choreography are the two

[1] Our use of the term "business transaction" in this paper refers to any unit of work carried out across microservices [181], not necessarily under ACID guarantees.

types of interactions that take place in the context of distributed web applications. Most papers [138, 111, 216, 14, 163, 167, 143] and open-source projects [243, 76, 220, 284, 232, 266, 267, 23] report the use of the choreography conversation pattern [120] through both synchronous and asynchronous event-based workflows.

In open-source repositories, event-based workflows are dominant, implying that updates and operations affecting other microservices are queued for asynchronous processing. This finding has led us to observe that microservice architectures indeed follow the BASE model [204], 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 [57, 48, 14]. Only one open-source project [215] adopts a saga-like orchestration. In any case, the options above are characterized by weak consistency models [20] 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 [89, 288, 179] 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 (Figure 3.2 and Table 3.2), which indicates a functionality dependence between microservices. However, consistent with recent discussions in the community [199, 114, 268, 112], 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.

O5. Distributed commit protocols do not enjoy popularity in microservices. The lack of effective and intuitive application abstractions may play a role in refraining practitioners from adopting such protocols [114, 112].

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, open-source repositories, along with a preliminary assessment of the survey with industry representatives.[2] Ta-

[2]By investigating the literature, we have found that the patterns for inter-microservice

Table 3.2: Mechanisms for inter-microservice coordination

Coordination mechanism	#	%
Orchestration	37	22.84
Sagas (centralized approach, with a Saga coordinator)	24	14.81
The Back-end for Front-end Pattern (BFF)	24	14.81
Choreography	22	13.58
Sagas (decentralized approach, i.e., no Saga coordinator)	14	8.64
Distributed transactions (e.g., via 2PC) [3]	14	8.64
2-Phase Commit	11	6.79
Others	16	9.88

ble 3.2 shows the responses sorted in descending order.

In contrast with findings from literature and open-source repositories, orchestration-like (including sagas [98] and the backed-end for front-end pattern (BFF)[4] [35, 258, 180]) 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. 32 out of 90 (35.5%) responded and most responses (78%) 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 (22.8%), chore-

operations are not homogeneously defined. Besides, the preliminary assessment confirmed that developers tend to mark this option in a specialized way, e.g., centralized sagas are infrequently identified as orchestration or decentralized sagas as choreography. We then opted to proceed with different options, even though some may be subclasses of others.

[3]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). We then opted to change the option to *2-Phase Commit*. By analyzing such answers, we estimate that 11 of the responses are not compatible with distributed transactions. This would result in 33 responses (20.37%) being attributable to choreography in total.

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

ographies are also highly mentioned (20.4%, see footnote 3). An interesting quote provided by one of the respondents characterizes how choreographies are implemented in industry, which aligns well with our findings in literature and open-source applications:

> 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 synchronous and HTTP-based, whereas the latter is mostly event-based and asynchronous.

A substantial fraction of the surveyed papers [167, 163, 216, 111, 57, 48, 260] employ choreography as the mechanism for cross-microservice coordination and do so by means of the implementation of asynchronous and event-based workflows. This pattern is consistent with the responses obtained from practitioners, as choreography is highlighted as one of the most prevalent mechanisms (Table 3.2). Most analyzed open-source repositories [76, 215, 220, 284, 267, 266, 23, 243] also show the same pattern.

O6. The results suggest the prominence of orchestration-like mechanisms in industry settings, in contrast to the prevalence of choreography in the open-source repositories. Additionally, 2PC is not used often, while asynchronous and event-based coordination is the norm.

3.4.2 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 [14, 143, 48, 260] and open-source repositories [215, 76, 232, 220, 284]. 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. 27 (40.3%) of the participants declared the use of some mechanism and 22 of them (81.4%) 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 HTTP requests, 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) Six respondents described the use of composition of service calls, i.e., a microservice performs the necessary synchronous requests to retrieve data from other microservices. (ii) One respondent declared the use of the BFF pattern [35]. We also identified such practice in a repository [76]. (iii) Lastly, one respondent declared the use of the API Gateway pattern [214]. We also observed its adoption in open-source repositories [215, 232, 284] and the literature [14, 138]. 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 purposes. 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. In 7 out of 9 (77 %) projects analyzed [284, 220, 23, 215, 76, 215, 243], we found code fragments used for communication-based replication that presuppose weak delivery semantics. In other words, although updates to the same object are often sequentially ordered by the publisher, there is no ordering guarantee regarding updates to different replicated objects. As a result, causal dependencies are ignored on updating replicated data items. The responses provided by 5 participants in open answers suggest the same trend. 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 [257]. 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. In other words, microservices can be easily put to and removed from operation while not affecting others, whereas such a pattern is often not found in computational steps of ETL tools.

(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 3.3.2), 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.

O7. The results highlight that the decentralized data management principle does not refrain microservices from performing queries over distributed states. As a result, practitioners often rely on *ad-hoc* mechanisms for data processing at the application level.

3.4.3 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. 18 (26.86%) out of 67 respondents declared the use of DSPSs and 13 (19.4%) of them provided a description. We summarize these as follows.

Data processing pipelines. We observed the use of application libraries targeted at stream processing [130] in microservices to perform data transformations, as mentioned by a 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 mismatch between microservices and DSPSs [131, 174, 265], 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 within this abstraction. The tight coupling of operators and their static dependencies contrasts with microservice principles, including loose coupling, fault isolation, independent evolution, and autonomous deployment. Most importantly, 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.

O8. 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.

3.4.4 Event-driven computing

As mentioned previously, we observed that microservice architectures often follow the BASE model [204], wherein organizing a computation in an event-driven architecture (EDA) is argued for scalability and architectural

decoupling. In an EDA, events that are relevant to an incoming request are generated when a consistent state is reached to allow further processing [204]. 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 provide an example of one of their use cases in a short answer. A total of 45 (67.16%) out of 67 participants indicated the use of such computations and 26 (38.81%) provided a brief use case description.

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."

3.5 Microservices through the Looking Glass

In Sections 3.3 and 3.4, we observed that microservice applications deviate significantly from the architecture of traditional database applications, introducing significant data management logic at the application level 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.

3.5.1 Cross-microservice validations

```
1 // Cart microservice: Checking availability of products
2 private boolean checkAvailableInventory(ShoppingCart cart) {
3     List allInventories = getInventoryEndpoint()
4             .HttpGet(cart.getProductItems());
5     return allInventories.map(inventories -> {
```

```
6           List insufficient = inventories.filter(item ->
                item.get("inventory") - item.get("amount") <
                0).toList();
7           if (insufficient.size() > 0) return false;
8           else return true; });
9   }
```

Listing 3.1: Cross-microservice validation example

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* [284] shown in Listing 3.1. 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 a pattern in several other projects, such as [215, 232, 243, 76, 23, 267].

Discussion. In the case presented in Listing 3.1, 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 [112], 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.

3.5.2 Implicit cross-microservice associations

```
// Product microservice: manages available products
public void deleteProduct(String productId,
    Handler<AsyncResult<Void>> resultHandler) {
    this.removeOne(productId,DELETE_STATEMENT,resultHandler);
}
-------------------------------------------------------------
// Cart microservice: product items without association
public class ShoppingCart {
  private List<ProductTuple> productItems;
  private Map<String, Integer> amountMap;
  public ShoppingCart() {}
} // additional code omitted
```

Listing 3.2: Implicit cross-microservice association example

Background. Even though microservices follow an approach similar to BASE for functional decomposition [204], developers do not refrain from modeling implicit associations across microservices.[5] We encountered several cases where enforcement of foreign key constraints across microservices is a necessary condition for ensuring correctness, and the applications analyzed show no evidence of such enforcement.

Consider the source code snippet exhibited in Listing 3.2 adapted from *event-stream* [23], *vertx* [284], and *eShopContainers* [76] applications. In the event of a product removal from the *product* microservice, in the absence of cascading delete, operations carried out by the system over records stored in other microservices that rely on the existence of the deleted product(s) will still assume such a product exists, which may bring the system to an inconsistent state. This pattern is observed across several other microservice applications [243, 23, 76, 284, 215, 267].

Discussion. Microservice developers have to explicitly enforce implicit foreign key constraints across microservices to avoid bringing the system into an inconsistent state, a complicated 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 leads to "orphaned" records in one or more microservices. Such a pitfall is even worse

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

than encoding associations under non-serializable isolation, since it may lead to a much higher number of anomalies [18].

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

3.5.3 Cross-microservice queries

Background. As part of our study, we observed the popularity of online queries in microservice architectures (Section 3.4). Given the distributed nature of data stores, the data encapsulation may introduce challenges not found in traditional monolithic systems.

The literature [14, 138] and open-source repositories provide interesting examples of microservices being employed for data aggregation through queries spanning different microservices (and their 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 [215], shown in Listing 3.3. 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).

```
1  public OrderDetails getOrderDetails(Long orderId) {
2      OrderInfo orderInfo = orderService.findOrderById(orderId);
3      TicketInfo ticketInfo = kitchenService
4              .findTicketById(orderId);
5      DeliveryInfo deliveryInfo = deliveryService
6              .findDeliveryByOrderId(orderId);
7      BillInfo billInfo = accountingService
8              .findBillByOrderId(orderId);
9      Mono<OrderInfo,TicketInfo,DeliveryInfo,BillInfo> combined =
           Mono.zip(orderInfo, ticketInfo, deliveryInfo, billInfo);
10     OrderDetails orderDetails = combined.map(
           OrderDetails::makeOrderDetails);
11     return orderDetails;
12 }
```

Listing 3.3: Cross-microservice query example

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 [19], which 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 [203] is not possible. Lastly, with such a sequential request pattern, as observed in Listing 3.3, some (or all) requests to microservices may fail, thus leading to missing records and an incomplete result. Such a pattern is also found in [76, 232, 243, 267].

C3. Developers have no support for querying multiple microservice database states consistently and they end up encountering challenges on reasoning about the application state.

3.5.4 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.

```
1 // Basket microservice: Basket checkout asynchronous request
2 public async Task CheckoutAsync(Checkout c_out,long userId){
3     var basket = await _repository.GetBasketAsync(userId);
4     var eventMessage = new CheckoutAcceptedEvent(userId,
          c_out.Buyer, c_out.RequestId, basket);
5     try { _eventBus.Publish(eventMessage); }
6     catch (Exception ex){
7         _logger.LogError(ex, "ERROR Publishing integration event:
              {IntegrationEventId}", eventMessage.Id);
```

```
8           throw;
9       }
10  }
11  ------------------------------------------------------------
12  // Catalog microservice: Update product asynchronous request
13  public async Task UpdateProductAsync(Item productToUpdate){
14      // code omitted
15      if (productPriceChanged){
16          var priceChangedEvent = new ProductPriceChangedEvent(
17              catalogItem.Id, productToUpdate.Price, oldPrice);
18          await _catalogEventService.
19              PublishThroughEventBusAsync(priceChangedEvent);
20      }
21  }
22  ------------------------------------------------------------
23  // Basket microservice: Reaction to ProductPriceChangedEvent
24  public async Task Handle(ProductPriceChangedEvent @event){
25      var userIds = _repository.GetUsers();
26      foreach (var id in userIds){
27          var basket = await _repository.GetBasketAsync(id);
28          await UpdatePriceInBasketItems(@event.ProductId,
                @event.NewPrice, @event.OldPrice, basket);
29      }
30  }
```

Listing 3.4: Feral ordering example

Consider the example adapted from the project eShopContainers [76] shown in Listing 3.4. 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 [76, 284, 243, 267, 23].

Discussion. Literature mentions that developers face challenges in specifying the consistency requirements for their applications [248]. Given that microservices comprise a class of applications that adopt a distributed architectural style [179], 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.

C4. 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.

3.5.5 Replication hell

Background. As mentioned in Section 3.4, 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 [203]. This pattern is observed in several microservice applications [215, 76, 243, 220, 267, 23].

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 [260] hold potential to provide more principled replication guarantees in microservices. Even though Synapse [260] is grounded on the industry-strength MVC pattern [96] 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.

C5. The lack of comprehensive support for data replication across microservices lead developers to rely on ad-hoc application-level replication mechanisms, a choice that often leads to inconsistency among microservice states.

3.5.6 Non-transactional queuing

Background. In an intra-microservice transaction, updates affecting other microservices are queued for asynchronous processing. As advocated by Pritchett [204], this queuing must be part of the transactional context of the database operation in the originating microservice. However, as observed in our extended version [145], there is no evidence of use of message persistence queues integrated with the database. Practitioners rely on external message persistence queues (e.g., message brokers) without employing a distributed commitment protocol for message queuing.

Consider the example adapted from the project Lakeside Mutual [243], shown in Listing 3.5. 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 [220, 243, 267, 23].

```
1  public ResponseEntity createInsuranceQuote() {
2     InsuranceQuote insuranceQuote = new InsuranceQuote
3        (date, QUOTE_SUBMITTED, insuranceOptionsEntity);
4     insuranceQuoteRepository.save(insuranceQuote);
5     policyMessageProducer.send(date, insuranceQuote);
6     return ResponseEntity.ok(insuranceQuote);
7  } // parameters omitted
8  ------------------------------------------------------------
9  public void send(Date date, InsuranceQuote insuranceQuote) {
10     InsuranceQuoteEvent insuranceQuoteEvent =
11        new InsuranceQuoteEvent(date, insuranceQuote);
12    try { jmsTemplate.send(insuranceQuoteEventQueue,
          insuranceQuoteEvent);
13    } catch(JmsException exception) {
14       logger.error("Failed to send", exception);
15    }
16 }
```

Listing 3.5: Non-transactional queuing example

Discussion. Existing database systems often provide support for persistent messaging or notification mechanisms integrated into the database [70]. However, to use these mechanisms, different 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 [204].

C6. 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 ensuring application correctness.

3.6 Delving into Developers' Pains

To strengthen our confidence on the practical challenges revealed in the last section and derive additional ones, we inquired 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 [145].

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, matching our hypothesis highlighted in **O2**. 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." An-

other 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. In line with open-source projects and the literature, respondents have also pointed out eventual consistency (15.68%) as a challenge. 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 3.5.5); (ii) employing workflow-oriented business transactions across microservices, often driven by asynchronous and event-based communication (Section 3.4).

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 multi-microservice workflow has terminated and what its current state is.

C8. Due to the distributed nature of microservice architectures, eventual consistency is often taken as the de facto consistency model by practitioners. This choice 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 requires 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 (§ 3.3.2) 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 (§ 3.4.2 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." Although initial work suggests considering database-related attributes, such as relationships between relations to derive a functional partitioning [144], further explorations with real-world systems are necessary.

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 [97]) fail to incorporate the asynchronous and event-driven properties of 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 from effectively experimenting with and reasoning about microservice deployments.

Summary. The perception of developers regarding the challenges faced while dealing with data management are very aligned with our findings discussed in § 3.5. Additional challenges were revealed, such as schema evolution (**C7**) and data cleaning (**C9**). This overall perception strengthens our confidence that there is an unaddressed need for effective data management support in microservices.

3.7 Towards Microservice-oriented Database Systems

In this section, we devise a candidate set of features for a microservice-oriented DBMS based on the observations and challenges identified in the previous sections. Then, we turn our attention to explain why state-of-the-art DBMSs are insufficient for the needs of microservices in the light of the proposed features. Finally, we discuss how to realize the features into a microservice-oriented DBMS.

Required features for a microservice-oriented DBMS.

As explained in § 3.5.4, developers have no way to specify event-based constraints. For example, concurrent checkout requests in eShopContainers [76] may be processed in arbitrary order wrt. other events, hence providing no guarantee of applying updated prices before checkout.

F1. The database system should be aware of events cutting across several microservices, so that it can capture the complex interplay of data exchange between microservices.

F2. Event-based awareness allows the database system to enforce event-based constraints, so that application safety can be supported.

Besides, as explained in § 3.3.1, § 3.4.1, and § 3.4.4, practitioners employ communication through asynchronous events to decouple microservices. However, it is observed that, by decoupling, developers have a hard time enforcing high data consistency (§ 3.5.4, § 3.5.5,§ 3.5.6).

F3. The database system should provide abstractions that strike a balance between loose coupling among components and high data consistency, so that effective and efficient cross-microservice coordination can be achieved.

Constraints cutting across several microservices are often enforced through application-level validations (§ 3.5.1). As explained in § 3.5.2, such an approach leads to an implicit logical data dependency – given that it is not formalized as a constraint – from the *Cart* microservice to the *Product* microservice.

F4. The database system should allow users to specify cross-microservice associations and enforce them consistently, so that users can avoid encoding error-prone validations at the application level.

In addition, as microservices extensively employ cache stores to reduce the costs of querying database states, the constraints across microservices should also take into account the consistency of the data stored in the underlying microservices' cache stores (**C9**).

F5. Database systems should allow the user to specify the freshness semantics between the underlying cache and database states, so that application safety can be safeguarded through adequate cache consistency management at the microservice level.

As explained in § 3.4.2, replication is often employed so that microservices can avoid coordination via cross-microservice queries. Microservices then queue or publish their own data items' updates as asynchronous events, which are then eventually delivered to consumer microservices. However, expressing data replication through application-level mechanisms creates a great barrier for developers.

F6. The database system should offer effective abstractions for users to specify data replication across microservices, so that varied replication consistency models can be properly supported.

In § 3.4.2 and § 3.6, we observed the practice of retrieving data from sev-

eral microservices to build a real-time view. However, microservice developers find challenges when implementing consistent cross-microservice queries, such as point-in-time views (§ 3.5.3).

F7. The database system should provide native support for consistent cross-microservice online queries, but at the same time respect the data sovereignty principle of microservices.

F8. The database system could optimize cross-microservice queries through pre-processing the data, pre-joining the data, or proactively replicating the data owned by different microservices while fulfilling the consistency-isolation levels required by the application.

Web-Scale Application Architectures not Enough.

Although microservice applications can be considered a realization of BASE, an approach where functional partitions are eventually consistent, the microservice paradigm introduces data management requirements not originally envisioned by Pritchett [204], including but not limited to application-level replication, consistent cross-microservice queries, cross-microservice validations, heterogeneous services, events cutting across several microservices, and interleaving of distinct, but correlated events that lead to data races. As prescribed by the BASE model, functional partitioning necessarily involves pushing constraint enforcement to the application level [204], which as we have seen often leads to consistency problems in microservice architectures (§ 3.5). Lastly, BASE can be considered a set of principles for designing functionally partitioned applications, thus not a full-fledged data management solution.

Furthermore, there have been several recent works in Internet-scale database services. These systems provide high availability at a global scale, at the same time offering high throughput data processing and multi-tenancy by design. While some of them support strong consistency guarantees, such as Google Spanner [54], Amazon Aurora [259], and Azure Cosmos DB [15], others trade stronger consistency for performance, such as DynamoDB [66] and Astra DB [241]. We address the common shortcomings of these solutions in the context of the aforementioned features.

F1 & F2. In addition to the lack of atomic event queuing in Internet-scale databases (§ 3.5.6), database systems are unaware of the complex interrelationships among events in microservice applications. Thus, it would be desirable for developers to be able to specify event-based constraints on

their processing, with the enforcement being performed at the database layer (§ 3.5.4).

Consider, for example, a scenario found in popular e-commerce microservice applications [76, 284, 266, 23] where an order checkout is optimistically processed, i.e., the products in the order are sold without checking their availability in stock. Connecting with Figure 3.1(b), the *Stock* microservice might subscribe to checkout and payment events in order to determine whether the process of acquisition of new stock should be triggered. However, the *Stock* microservice should only take into account checkout events whose corresponding payment events have arrived, since the sale of the products is only finally confirmed after payment.

In the presented scenario, resorting to a single tenant in a multi-tenant database system would allow practitioners to write database triggers for the latter functionality; however, this approach introduces an undesirable dependence between the schemas of the various microservices. By contrast, practitioners could resort to multiple tenants, e.g., one per microservice for isolation. Unfortunately, in this case, the checking of the stock replenishment condition would need to be pushed to the application level.

F3. As pointed out above, it is not obvious how to employ a multi-tenant DBMS to store the states of distinct microservices. One solution is to employ one tenant per microservice, but this leads the states to be completely isolated. Another possibility is to use a single tenant for the states of multiple microservices; however, the latter implies complete sharing. For instance, Figure 3.1(b) depicts the case where Orders requires coordinating with Stock, Campaigns, and Discounts to safeguard that an order transaction is correctly placed. Coordinating these 4 microservices would require either resorting to application-level 2PC or ad-hoc application-level coordination mechanisms (**O5**, **O6**) – in case states were partitioned across tenants – or use of DBMS transactions – in case states were shared. In either case, the principles of loose-coupling and autonomy among microservices would be jeopardized. On the one hand, application-level 2PC would imply API coupling with high consistency, while ad-hoc application-level coordination would typically entail loose coupling at lower data consistency levels. On the other hand, sharing of the same database tenant would imply that microservices lose the ability to independently evolve and fail.

F4 & F5 & F6. In the same line of reasoning of **F3**, multi-tenant database systems (i) cannot enforce cross-microservice constraints, (ii) cannot reason about cache consistency over distinct microservices' database states, and (iii)

cannot provide consistent data replication across tenants without failing to meet the decentralized data management and data sovereignity principles prescribed by the microservice paradigm. Consider, for example, the case where the state of *Payment* microservice refers to the customers relation in the state of the *Customer* microservice. By resorting to a single tenant for both microservices, we could potentially express a foreign key across their respective schemas. However, that would create a schema and failure dependency between the microservices. By resorting to different tenants, there would be no way to express the foreign key constraint across the two microservice states.

F7 & F8. Consider a scenario where in the context of a customer's complaint about an undelivered package, a front-end microservice may need to provide a real-time view over the customer's interactions, discounts being applied, and the customer's credit score to enable business users to decide whether the customer can receive an additional compensation for what happened or only be reimbursed. Such an operation necessitates querying data from several microservices' states. However, it is unclear how existing database systems can support consistent cross-microservice queries.

Simply providing querying services over the private states of multiple tenants is insufficient to achieve consistent views. Rather, the DBMS needs to capture the interactions between the microservices so that it can analyze and support consistency semantics. Although a DBaaS vendor could potentially provide a holistic distributed state view across tenants (considering all microservice states are deployed as a tenant), such a feature would still be insufficient if not addressed in conjunction with all the prescribed features that necessarily require knowing more about the application, as we address next.

On realizing features into a microservice-oriented DBMS.

Usually, data management tasks executed at the application level are a black-box to the database system [271]. As a result, the database system is often unaware of the complex data management logic and constraints put forth by application developers [18]. This impedance mismatch is particularly worsened when it comes to microservices, since data is flowing outside of the database [112], that is, there are many data management tasks that cut across microservices. The DBMS is oblivious to these tasks. To realize the features described, we believe it is essential for the database system to better understand what is occurring at the application level. To bridge this gap, we envision two paths:

(A) Extending current abstractions for building stateful middle-tier applications can be seen as a fruitful solution. Laudable initial efforts on virtual actors [25] and stateful functions [245] have not yet incorporated the features required by microservice applications to a sufficient degree. For example, in Orleans [27], data durability functionality is explicitly managed by the developer. In addition, there is no way to specify event-based constraints, and it is unknown whether all microservice applications can be modeled through virtual actors [190, 265].
(B) Extending the frontiers of a database system by having database components living in the application tier and communicating the application's needs to the database system. With current database APIs, applications have no way to push down complex dataflows to the database system in a meaningful way, because interfaces abstract away the complex interaction and data exchanges happening outside the database. Although an initial proposition in this line has been presented in [149], no formalism and implementation has been provided so far.

F9. Appropriate abstractions should be formalized to allow the database system to gain knowledge of the data management logic carried out by the application and the complex interplay of microservices. These abstractions could help in achieving a proper division of tasks between the application and the database system, where data management tasks are pushed to the database system, thus alleviating the burden on developers.

3.8 Conclusion

In this paper, we observe that the lack of a holistic data management solution for microservices leads practitioners to resort to *ad-hoc* designs. These are characterized by weaving together heterogeneous services and resorting to application-level data management mechanisms, which lead to problems that cannot be resolved through code refactorings [86]. To tackle the data management requirements of microservices by design, we present a set of candidate features for DBMSs so that they can play a central role in this new paradigm.

Chapter 4

Online Marketplace: A Benchmark for Data Management in Microservices

Microservice architectures emerged as a popular architecture for designing scalable distributed applications. Although microservices have been extensively employed in industry settings for over a decade, there is little understanding of the data management challenges that arise in these applications. As a result, it is difficult to advance data system technologies for supporting microservice applications. To fill this gap, we present Online Marketplace, a microservice benchmark that incorporates core data management challenges that existing benchmarks have not sufficiently addressed. These challenges include transaction processing, query processing, event processing, constraint enforcement, and data replication. We have defined criteria for various data management issues to enable proper comparison across data systems and platforms. After specifying the benchmark, we present the challenges we faced in creating workloads that accurately reflect the dynamic state of the microservices. We also discuss implementation issues that we encountered when developing Online Marketplace in state-of-the-art data platforms, which prevented us from meeting the specified data management requirements and criteria. Our evaluation demonstrates that the benchmark is a valuable tool for testing important properties sought by microservice practitioners. As a result, our proposed benchmark will facilitate the design of future data systems to meet the expectations of microservice practitioners.

4.1 Introduction

Microservice architecture has emerged in the last decade as a popular architectural style in industry settings. This style promotes the decomposition of an application into independent microservices with associated private states. From an organizational point of view, these principles allow different teams to manage and evolve their own modules independently. At the same time, it enables new modules to be introduced and deprecated modules to be removed without impacting the application as a whole. From a technological point of view, each module can be independently deployed on distributed computational resources, allowing for failure isolation and high availability. Meanwhile, the message-based communication paradigm serves as a powerful abstraction for triggering tasks in remote microservices, facilitating data replication among microservices, and enabling failure recovery by replaying past events [133].

This industrial popularity has prompted cloud providers to offer rich features targeting microservice deployments [49, 16, 50, 13], such as specialized container-based technologies to deploy and scale microservices, message brokers to allow for loose-coupled microservices, multi-tenant DBMS technologies to meet the isolation properties of microservices, and application frameworks and side-car technologies to speed up deployment, code evolution and maintenance of microservices.

Despite the benefits of the decoupled design, a recent study [150] demonstrates that practitioners encounter several challenges when trying to meet data management requirements in this architecture, including distributed transaction processing, data replication, consistent data and event querying and processing, and enforcing data integrity constraints. In short, microservices are designed to function independently, but in practice, they often rely on each other's data and functionality to complete a workflow. As a result, it is essential to benchmark microservices in a way that accurately reflects the needs of practitioners. Unfortunately, existing microservice benchmarks do not fully capture these real-world requirements [150, 97, 285]. For example, DeathStarBench [97] and TrainTicket [286] do not consider event processing, nor do they specify what data invariants and transactional guarantees are necessary. The absence of a comprehensive benchmark for measuring the performance of data management tasks in microservices leads to the development of custom benchmarks that fail to reflect the complexities of data

management in real-world microservices [31].

To bridge this gap, we propose *Online Marketplace*, a novel microservice benchmark containing eight microservice types, ten event types, seven query types, including read-only and read-write queries, that reflect the key data management tasks pursued by practitioners, such as distributed transaction processing, data replication, consistent queries, event processing, and data constraint enforcement. To reflect the data management challenges mentioned above, we prescribe seven data management criteria that a data management platform should meet, including functional decomposition, resource isolation, data consistency, and data integrity of microservices. These criteria are meant to embrace the complex nature of deployments found in industry settings and facilitate conducting a fair comparison between different systems on the same basis. To our knowledge, *Online Marketplace* is the first microservice benchmark that embraces core data management requirements sought by microservice practitioners.

Based on the definition of *Online Marketplace*, we further developed a data generator and a benchmark driver. The latter can continuously generate and submit transactions to an implementation of *Online Marketplace*. A challenge of implementing the driver is generating transactions at runtime that coherently reflect the dynamic application state, for example, the latest product prices and product versions. Querying the runtime application state imposes an unnecessary and prohibitively expensive workload on the system. To address this problem, we developed a stateful driver, which manages a consistent mirror of some application data and generates coherent transaction inputs.

We verify the applicability of our benchmark by implementing five versions of *Online Marketplace* in three state-of-the-art data platforms, Orleans, Statefun, and a composite solution. Orleans and Statefun are designed for event-driven, distributed stateful applications. The composite solution reflects a usual architecture seek in practice [150, 133]. While implementing *Online Marketplace* on these platforms, we encountered several limitations that prevented us from fulfilling some of the data management functionalities and criteria sought in practice. We conducted experiments to measure how these competing platforms perform under different workloads and observed their design and architectural implications in performance. Our results show that our benchmark can effectively stress the performance of the platforms and reveal performance and functionality issues. Meanwhile, through *Online Marketplace*, we derive actionable insights to foment the design of futuristic

data systems that will meet the expectations of microservice practitioners, who are an important part of the database user community.

By providing a concrete example for database researchers to understand the unmet needs of data management in microservices, the benchmark serves as a testbed for experimenting with novel algorithms, approaches, techniques, and programming models. For instance, a recent study [156] aims to fulfill the data management criteria outlined here to enhance state management features in actor systems for cloud-native applications. Furthermore, Online Marketplace is a benchmark that not only benefits the real-world microservice applications that served as inspiration for our design [103, 272, 226, 47, 80, 81, 218, 6, 132, 46, 224, 225, 242, 166, 183, 52, 260] but can also extend beyond existing microservices applications. For instance, besides microservices, there are trends of other architectural styles and programming models offering trade-offs between complexities in data management and the degrees of coupling. These include modular monolithic architectures, FaaS, Actors, etc. As demonstrated in our case studies (§ 4.7), *Online Marketplace* is well positioned to serve as a testbed for these trending cloud application architectures and programming models. The artifacts used in this work are available online [140] for reproducibility and extension.

The Online Marketplace Benchmark is available as an open source repository: `https://github.com/diku-dk/OnlineMarketplaceBenchmark`. The repository contains the driver and references to several benchmark implementations, including Microsoft Orleans [34], Apache Flink Statefun [239], and Microsoft Dapr [171].

4.2 Background

Data Management Challenges. Despite more than a decade-long employment in the industry, little work in data management research has incorporated the challenges brought about by data-intensive applications following the microservices architectural style. Based on the findings of a recent study that empirically investigated the data management challenges in microservices [150] and public reports [103, 272, 226, 47, 80, 81, 218, 6, 132, 46, 224, 225, 242, 166, 183, 52, 260] of companies with large-scale deployments, we summarize the core data management challenges as follows:

(i) **Ensuring all-or-nothing atomicity.** The asynchronous and non-blocking nature of messages and the lack of interoperability across different

data stores make distributed commit protocols difficult to implement, leading developers to either encode their own or eschew the use of synchronization mechanisms.

(ii) **Implementing efficient and consistent data processing.** While data is scattered across microservices, some workloads require querying and joining data belonging to different microservices. Therefore, developers often need to implement querying functionalities at the application layer that should belong in the database layer.

(iii) **Ensuring data replication correctness.** In order to reduce the expenses associated with querying data from remote microservices, developers often resort to caching or replicating data by subscribing to events generated by other microservices. However, as these events can arrive in any order, developers face challenges in maintaining consistent replication.

(iv) **Enforcing cross-microservice data integrity constraints.** As the application is functionally partitioned, data integrity constraints can span multiple microservices. This creates major challenges in constraint enforcement.

(v) **Ensuring correct event processing order.** Developers leverage application-generated events to implement event-based microservice workflows. However, guaranteeing the correct processing order can be challenging due to the asynchronous nature of events. Events can arrive out of order, late, or even duplicated. Such scenarios pose a significant issue when the application logic is sensitive to the processing order of events.

Problems of Existing Benchmarks. Existing microservice benchmarks like DeathStarBench [97] and TrainTicket [286] do not fully capture these real-world challenges, as shown in Table 4.1. DeathStar [97] was created to investigate the effects of microservice architectures on hardware and software in system stacks, particularly network and operating systems. TrainTicket focuses on replicating industrial faults in microservices, supporting researchers on investigating fault analysis and debugging. However, they do not take into consideration asynchronous events; thus, challenges related to correct event processing are missing. Furthermore, they do not consider any requirements on data invariants, transactional guarantees, data replication or data querying. Therefore, they target problems that are either oblivious to data management or position data management as not a primary concern.

Similar to microservices, Object-Oriented DBMSs (OODBMS) also adopt the modularization principle, such as encapsulating state and operations into objects within the database. Although, in principle, a microservice can be

Table 4.1: Comparison of microservice benchmarks

Requirement	**Criteria**	**DeathStar & TrainTicket**	**Online Marketplace**
Functional Decomposition	Isolation of Resources	Yes	Yes
Asynchronous Events	Event Processing Order and Delivery Guarantees	No	Yes
Event Processing		No	Yes
Distributed Transactions	All-or-nothing Atomicity; Isolation Levels Allowed	No	Yes
Data Invariants	Data Invariant Enforcement	No	Yes
Data Replication	Data Caching and Replication Consistency	No	Yes
Query Processing	Query Processing Consistency	No	Yes

mapped to an object in OODBMS, we find that OODBMS benchmarks cannot meet the needs that we target. Benchmarks like Cattell, HyperModel, OO7, Justitia, and OCB aim for modeling object relationships in the context of engineering applications and evaluating object clustering algorithms [62]. All these fail to capture the data management requirements in Table 4.1.

TPC-W [56] is a transactional benchmark that models the core aspects of user experience on an e-commerce website, such as browsing pages, checking out books, and searching for keywords. TPC-C [55] was designed to reflect the transaction processing of a wholesale supplier. YCSB [53] models transactional workloads in the cloud that are not necessarily executed under ACID semantics. All of them assume a traditional monolithic application architecture. Our benchmark models a modern microservice-oriented application and the complex interplay of their components through events, differing substantially from these benchmarks regarding architectural style and data management requirements.

Unibench [277] offers OLTP and OLAP workloads for multi-model databases.

However, it does not model the decomposition of microservices, which results in the absence of distributed and encapsulated states as found in microservice applications. On the other hand, stream processing benchmarks like Linear Road [7] focus on modeling continuous and historical queries, but they do not include transactional workloads.

Therefore, existing benchmarks cannot be used to benchmark the overhead of addressing the correlated data management challenges, which includes the overhead of all-or-nothing atomicity, concurrency control, consistent query processing, consistent data replication, and correct event processing. Consequently, they cannot be used to facilitate the development of new data systems to meet the real-world data management challenges of microservices.

4.3 The Online Marketplace Benchmark

4.3.1 Design Goals

To bridge the aforementioned gap, it is key to pick an application scenario that covers all the aforementioned challenges. We find online marketplaces meet the following design goals.

Industry strength. Online marketplaces present a substantial user base over the world [240]. It is widely reported that online marketplace platforms employ microservice architectures to achieve benefits such as loose-coupling, fault tolerance, workload isolation, higher data availability, scalability, and independent schema evolution [40, 178, 263, 30].

Low reproducibility barrier. In the same spirit of the design of TPC-C [55], online marketplaces fall into an application domain that can be easily understood by non-domain experts. This, in turn, reduces the learning curve and encourages benchmark adoption.

Generalizability. Many other popular application domains can be mapped to an online marketplace. These include multi-tenant e-commerce platforms, airline retailing [124] (e.g., hotels, travel agencies, and taxi and insurance companies as sellers), social media marketplaces, delivery service (e.g., connecting individuals to a myriad of businesses, like restaurants), digital investment (e.g., funds from different banks as products), and homestay platforms. It is noteworthy that many of these applications report data management challenges that we address in this work: event ordering and data

invariants [81, 218, 272, 226, 46, 224, 225] transactional consistency across microservices [103, 132, 47, 80, 242], consistent data querying [166, 183, 52] data replication correctness [260, 218, 6].

In sum, an online marketplace is a suitable use case to benchmark data management in microservices.

4.3.2 Application Scenario

Cart Management. Customers shop in *Online Marketplace*by navigating and selecting products from a catalog. A customer session is linked to a *cart*, with operations involving adding, removing, and updating items (e.g., increasing the quantity). A customer can only have one active cart at a time. When requesting a checkout, the customer must include a payment option and a shipment address, which are assembled together with the cart items and submitted for processing. On the other hand, customers may also abandon their carts before submitting the checkout request.
Catalog Management. Each product is offered by a particular seller. Sellers are responsible for managing their products and associated stock information. They may replenish products and adjust their prices.
Customer Checkout. Checkout requests follow a chain of actions: stock confirmation, order placement, payment processing, and shipment of items. Upon submission of an order, if one or more items do not have sufficient stock, the checkout will still proceed with the available items.
Payment Processing. Upon stock confirmation, the payment details provided by the customer are used to make a payment request to an external payment service provider (PSP) [126]. A payment can fail for two reasons: rejection of the transaction by the provider or impossibility of contacting the payment provider.
Order Shipment. Upon approval of payment, the shipment process starts. For each seller present in an order, a shipment request is created. A shipment request includes the items present in the order. Each item is reflected as a package that should be delivered to the customer at some point.
Package Delivery. Whenever a package is delivered, both the corresponding customer and seller are notified. When all packages of an order are delivered, the order is considered completed.

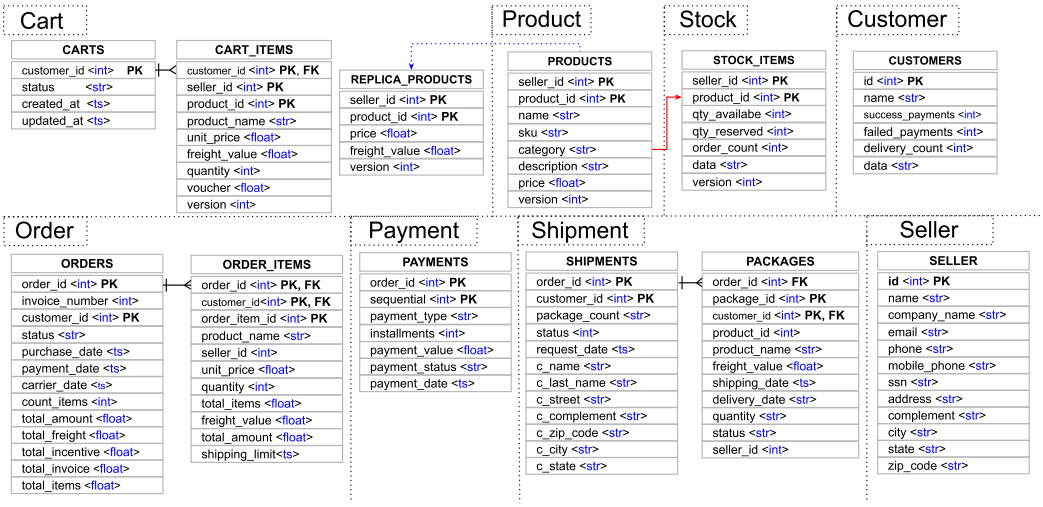


Figure 4.1: *Online Marketplace* Data Model

4.3.3 Microservices

Traditionally, a microservice-based application is often made up of independent services, each deployed in a container on an OS-level virtualized platform such as Docker [150]. However, the emergence of programming frameworks for designing distributed applications, such as Orleans [34] and Statefun [239], provides programming models that allow microservices to be implemented using higher-level abstractions, such as actors and functions, respectively.

To allow this benchmark to support the gamut of microservice implementations, we withdraw ourselves from defining an architectural blueprint, but rather, we focus on describing the expected independent components that must compose the benchmark application. In this sense, in the following paragraphs, we take advantage of Table 4.2, Table 4.3 and Figures 4.1 and 4.2 to clarify the interactions, APIs, and the data model prescribed for each microservice and event type. It is worth noting that we use the term "events" as a communication abstraction for microservices, but these can also be framed as any message payload asynchronously delivered to a microservice. Besides, the event identifiers in Figure 4.2 do not imply a particular order in which the microservices exchange events. Furthermore, while we specify the microservices' state and the queries using the relational model, they can also be specified through other data and query models as long as the same functionalities can be achieved.

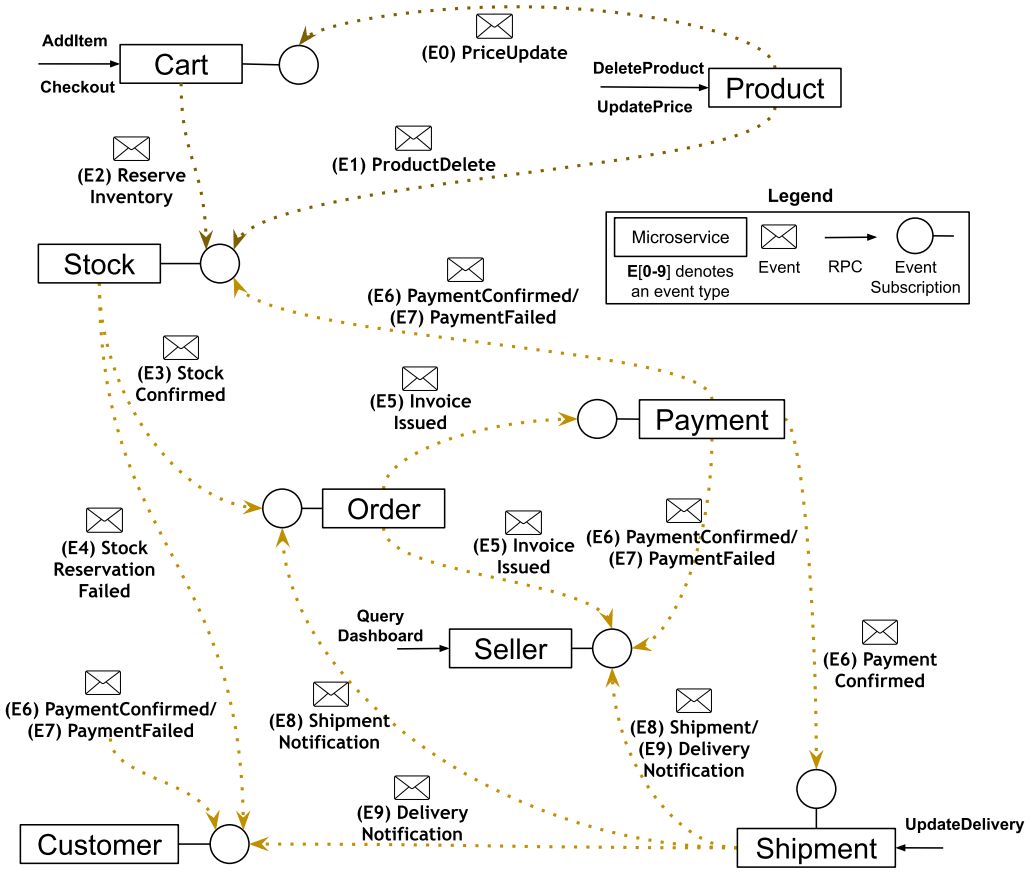


Figure 4.2: *Online Marketplace* Events

Table 4.2: Events Processed by *Online Marketplace*

Event Type	**Event Schema**	**Producer**	**Consumers**
`Price Update`	`seller_id, product_id, newPrice, version`	Product	Cart
`Product Delete`	`seller_id, product_id, version`	Product	Stock
`Reserve Inventory`	`checkout_info {customer_id, payment_type, card_number,...}, items: [{seller_id, product_id, price, version},...], timestamp`	Cart	Stock
`Stock Confirmed`	Same as `Reserve Inventory`	Stock	Order
`Stock Reservation Failed`	Same as `Reserve Inventory`	Stock	Customer
`Invoice Issued`	`order_id, invoice_num, total_amount, customer_info {customer_id, payment_type, card_number,...}, items: [{seller_id, product_id, price, version},...], timestamp`	Order	Payment, Seller
`Payment Confirmed`	Same as `InvoiceIssued`	Payment	Order, Seller, Shipment
`Payment Failed`	Same as `InvoiceIssued`, but including failure code	Payment	Order, Seller
`Shipment Notification`	`order_id,invoice_num,shipment_id, shipment_status,timestamp`	Shipment	Order, Seller
`Delivery Notification`	`order_id,invoice_num,shipment_id, package_id,seller_id, product_id,delivery_status,timestamp`	Shipment	Seller, Customer

Table 4.3: Required HTTP APIs

API	**HTTP Request Type**	**Microservice**	**Description**
/cart/{customerId}/add	PUT	Cart	Add a product to a customer's cart
/cart/{customerId}/checkout	POST	Cart	Checkout a cart
/cart/{customerId}/seal	POST	Cart	Reset a cart
/customer	POST	Customer	Register a new customer
/product	POST	Product	Register a new product
/product	PATCH	Product	Update a product's price
/product	PUT	Product	Replace a product
/seller	POST	Seller	Register a new seller
/seller/dashboard/{sellerId}	GET	Seller	Retrieve seller's dashboard for a given a seller
/shipment/{tid}	PATCH	Shipment	Update packages to 'delivered' status
/stock	POST	Stock	Register a new stock item

Cart. The *Cart* microservice allows the customer to manage the products that ought to compose a checkout order. It does so through APIs for managing cart items and submitting cart checkouts.

To manage a customer's cart, two relations are used: `carts` to track the status of a customer session, and *cart_items* to store the items that customers want to buy. Moreover, the *replica_products* relation represents a partial replica of the *product* table belonging to the `Product` microservice. The table is only updated through `ProductUpdate` events received from the `Product` microservice (represented by a dotted blue arrow in Figure 4.1). The replication semantics are discussed in Section 4.4.3.

When a checkout request is received, `Cart` ensures the correctness of the customer's cart by matching the cart items to the replicated product information and looking for products that had their price updated. In this case, the price is updated and the difference (if positive) between the new and the old price becomes a discount in the cart item. The customer's cart items are then assembled into a ***`ReserveInventory`*** event and published for asynchronous processing. Once the event is published, the cart is sealed, and the customer is able to initiate a new cart session.

Product. The *Product* microservice manages the catalog of products. It performs operations over a single relation: `product`. For every update operation triggered via the following API, a corresponding event is generated for downstream processing: E0 for UpdatePrice and E1 for DeleteProduct.

DeleteProduct. Marks a product as unavailable to customers.

UpdatePrice. Update the price of a product.

Stock. The *Stock* microservice manages the inventory through a single relation: *stock_item*. To update the inventory, the *Stock* microservice processes four events: `ReserveInventory`, `PaymentConfirmed`, `PaymentFailed`, and `ProductDelete`. Every inbound event leads to updates in one or more *stock_item*' tuples (depending on the number of items present in the checkout).

On processing a `ProductDelete` event, the *Stock* microservice marks the respective *stock_item* as unavailable, meaning that future checkouts (triggered by `ReserveInventory` event) on this item cannot be carried out (represented by a red connector in Figure 4.1).

Processing the `ReserveInventory` event leads to marking the stocks of some items (subject to availability) presented in the customer's cart as reserved. The result of processing `ReserveInventory` might lead to the generation of two events: (i) `StockConfirmed` if at least one product has

been reserved, and; (ii) `ReserveStockFailed` if no product requested by the customer is available. In both cases, the respective items are assembled together in the respective event for downstream processing. Lastly, `PaymentConfirmed` confirms the reservation and `PaymentFailed` would lead to withdrawing the corresponding stock reservations.

Although at first sight, the differences between Product and Stock may seem blurry, it is important to highlight that they provide distinct functionalities in the application. While the `Product` is responsible for the correctness of the catalog data, including but not limited to the characteristics of a product like name, category, seller, and price, the `Stock` microservice ensures the integrity of the inventory. It is important to address that such design reflects real-world deployments [244].

Order. The *Order* microservice manages customer orders' data. It does so by maintaining four relations: *order*, *customer_orders*, *order_items*, and *order_history*. The *order* relation contains general data about an order, including but not limited to customer, amounts, etc. *customer_order* relation tracks the number of orders requested by each customer, which is used to form the invoice number. Order items relate to the products requested in checkout and *order_history* tracks updates to an order (invoiced, paid, shipped, completed). Order's relations are updated upon processing the following events: `StockConfirmed`, `ShipmentNotification`, `PaymentConfirmed`, `PaymentFailed`.

Upon receiving a `StockConfirmed` event, the amount to charge the customer is calculated (including freight and discounts) and an invoice number is generated, resulting in a `InvoiceIssued` event. During this processing, all four relations have tuples created (in case it is the first customer order, otherwise *customer_order* tuple is updated). Afterwards, there are two moments where a customer's order status is updated: after a payment and after shipment updates. Both trigger writes to order and *order_history* relations. In this case, the order of event processing here plays a role on maintaining the notion of time progress for individual orders.

Payment. The *Payment* microservice is responsible for managing payment data. Two relations are used to track an orders' payment data: *payment* and *card_payment*. Payment processing starts through processing an `Invoice Issued` event. Every credit applied to the order, namely, discounts and the payment option chosen by the customer at the time of the checkout (credit/debit card or a bank slip) are stored in *payment* relations and in case of a card payment, a *card_payment* tuple is also inserted.

As part of the payment processing, it may be necessary to coordinate

with an external payment provider to ensure the provided payment method is valid. In this sense, *Payment* microservice relies on an idempotent API offered by the external payment provider. In case the payment microservice fails right after confirming a payment, subsequent attempts (after recovering from failure) to charge the customer does not incur in duplicate payments.

Processing an invoice can lead to two outbound events: `PaymentConfirmed` and `PaymentFailed`. The `PaymentConfirmed` event contains all items present in the paid invoice whereas the `PaymentFailed` is a simple payload containing the invoice number so the *Order* microservice can update its state accordingly.

Shipment. The *Shipment* microservice manages the lifecycle of a shipment processing, including shipment provision and delivery of goods. Two relations are maintained: *shipment* and *package*.

A shipment process starts by processing a `PaymentConfirmed` event, creating a delivery request for each order item and confirming all goods have been marked for shipment by producing a `ShipmentNotification` with status 'approved'.

At a later moment, through the *UpdateShipment* API, a set of shipments and associated packages tuples are updated. Each package delivered leads to the generation of a respective `DeliveryNotification` and, in case all items that form a shipment are delivered, a `ShipmentNotification` with status 'concluded' is also emitted.

Customer. The *Customer* microservice manages customer data, including their home address, contact information, and credit card. Besides, statistics about the customer are updated through processing the following events: `PaymentConfirmed`, `PaymentFailed`, `ReserveStockFailed`, and `Delivery Notification`.

Seller. The *Seller* microservice manages seller-based marketplace data. It does so by processing events and transforming them into seller-centric data. Three relations are found in *Seller's* schema: *seller*, *order_entry*, and *order_entry_details*.

The relation *order_entry* represents an order item but from the perspective of a seller. In this sense, amounts related to the item (i.e., discount, freight, total) are calculated. Every order (derived from the `InvoiceIssued` payload) is transformed into *N order_entry* tuples, where N is the number of items corresponding to a seller. The relation *order_entry* is updated through processing the following events: `PaymentConfirmed`, `PaymentFailed`, `Shipment Notification`, and `DeliveryNotification`.

4.3.4 Workload

In this section, we describe the *Online Marketplace* workload, namely, the business transactions and continuous queries the application must cope with.

4.3.4.1 Business Transactions

To realize the application scenario, we describe four business transactions that reflect different complexities in terms of the number of microservices involved and the number of events processed. In Section 4.4, we discuss how transactions possess different properties.

Customer Checkout. It starts in *Cart* microservice through processing a `Checkout` request. It involves *Cart*, *Stock*, *Order*, *Payment*, *Shipment*, *Seller*, and *Customer* microservices. A success path involves the following events: E2 - E3 - E5 - E6 - E8. There are two error cases: (i) when a payment fails, then from E5 we have E7 as the last; and when no single item from the cart is available in stock: E2 - E4. Although not shown in Figure 4.2 due to space constraints, a success path also involves E6 being processed by *Order*.

Price Update. To enable the partial replication of products in the *Cart*, upon processing a `UpdatePrice` request and updating its private state accordingly, the *Product* microservice generates E0 and sends it to the *Cart* microservice. By processing E0, the *Cart* microservice applies the new price to its corresponding replica.

Product Delete. To simulate making a product unavailable to customers, we pick a seller and a corresponding product (both from a distribution) and we set the product as disabled. To maintain the total number of products, thus avoiding anomalies in the distribution, we replace the deleted product with another one. This operation is done by processing a `DeleteProduct` request in *Product* microservice. Upon updating its private state accordingly, *Product* generates E1 and sends it to *Stock*.

Update Delivery. To simulate the delivery of goods, we pick the first 10 sellers with uncompleted orders (i.e., at least one package has not been delivered yet) in chronological order, and we set their respective oldest order's packages as delivered. Thus, packages are progressively delivered as more update delivery transactions are submitted to the system. For maintenance of statistics about customers and orders, the following events are generated with the transaction: `E9` is generated by *Shipment* for every package delivered and sent to the Seller and Customer microservices; and `E8` is generated by

Shipment when all packages of a shipment are delivered and sent to *Order*.

4.3.4.2 Continuous Queries

Continuous queries over event streams constitute an emergent trend in microservice applications. As event payloads produced by different microservices often contain state information [93], it is possible to (indirectly) access data from multiple microservices without breaking their encapsulation. In other words, continuous queries can be built based on events without resorting to (synchronously) pulling data from each required microservice.

In this section, we describe three different types of continuous queries covering important concerns in *Online Marketplace*. To specify the queries, we use the syntax of Materialize [127], a streaming database. Due to space constraints, we explain one continuous query and the others can be found in our extended version [145].

Query #1: Seller Dashboard. The business scenario encountered in a marketplace requires sellers to have an end-to-end overview of the operation in real-time to identify trends, such as the popularity of products, and to support decision-making, such as when to increase product prices.

Query Description: In this query, we want to determine the total financial amount of ongoing orders by seller. Assuming there is a stream (Listing 1) that filters out concluded and failed orders, one could implement this continuous query as shown in Listing 2.

Listing 4.1: Ongoing order entries base query

```
1 CREATE MATERIALIZED VIEW order_entries
2 SELECT order_id, seller_id, product_id, ...
3 FROM InvoiceIssued AS inv
4 LEFT JOIN ShipmentNotification AS ship ON ship.order_id =
      inv.order_id
5 LEFT JOIN PaymentFailed AS pay ON pay.order_id = inv.order_id
6 WHERE ship.status != 'concluded' AND pay.order_id IS NULL
```

Listing 4.2: Ongoing orders aggregation per seller

```
1 SELECT seller_id, COUNT(DISTINCT order_id) as count_orders,
     COUNT(DISTINCT seller_id, product_id) as count_items,
     SUM(total_amount) as total_amount, SUM(freight_value) as
     total_freight, SUM(total_items - total_amount) as
```

```
    total_incentive, SUM(total_invoice) as total_invoice,
    SUM(total_items) as total_items
FROM order_entries
GROUP BY seller_id
```

In addition, the seller dashboard output must also discriminate the records that compose the aggregated values computed in the above query. In this sense, it should also present to the user the following query result:

Listing 4.3: Ongoing orders discriminated

```
SELECT * FROM order_entries
WHERE seller_id == <sellerId>
```

Query #2: Cart Abandonment. A popular use case arising from a series of customer interactions is cart abandonment [164]. A cart is considered abandoned in two cases: (i) prior to checkout submission, and (ii) upon a failed payment processing, if no customer checkout re-submission is identified. **Query Description:** In this query, we want to find the cart checkouts that have either failed via stock reservation or payment attempt and have not been involved in a new checkout attempt within the next 10 minutes after the failure. Upon detecting an abandoned cart, a `CartAbandoned` event is generated for both *Cart* and *Customer* microservices.

Assuming there is a stream that is the union of both failed payments and failed reservations, one could implement **Cart Abandonment** in ksql [130] through the following continuous query:

```
CREATE STREAM failed_checkouts AS
SELECT cart_id, ts
FROM checkout_cart AS c
INNER JOIN failed_events ON cartid

SELECT failed_checkouts.cartid
FROM failed_checkouts AS fc
LEFT JOIN checkout_cart AS c
WITHIN 10 minutes ON stream1.cartid = stream2.cartid AND
    stream2.ts > stream1.ts
WHERE stream2.cartid IS NULL
EMIT CHANGES;
```

Query #3: Low Stock Warning. To assist sellers, marketplaces usually monitor products' stock proactively to notify sellers about low inventory and

ultimately refrain customers from experiencing unavailability of products.
Query Description: In this query, we want to find the products (and their respective sellers) that are likely to face unavailability in case no replenishment is provided in the near future. The search is based in the following criteria: The average number of items requested per week in the last month, independently of the result of the reservation and payment, is higher than the present stock level. The continuous query to warn sellers about low stock can be specified through the following statements in ksql:

```
CREATE STREAM qty_orders_per_week
SELECT productid, ts, sum(quantity) AS sum
FROM checkout_item AS c
GROUP BY productid
SLIDING WINDOW 1 week

CREATE STREAM agg_checkout
SELECT productid, sum(quantity)/count(*) as avg
FROM qty_orders_per_week AS q
GROUP BY productid
SLIDING WINDOW 1 month

CREATE STREAM stockthreshold
SELECT productid, avg
FROM agg_checkout AS c
INNER JOIN memory_stock_table AS mem ON productid
WHERE mem.quantity < avg
EMIT CHANGES;
```

4.3.5 Wrapping it Up

The application represents a real-world Marketplace scenario and the clients (Customers, Sellers, and Delivery Company) that interact with. Similarly to the state of the practice [221, 150], the application provides a balance in terms of how events are produced and processed, and how data is processed across different microservices. For example, while *Cart* and *Product* react to no events, acting as user-facing services by responding to synchronous requests, *Customer* and *Seller* microservices only react to events, producing none. Besides, *Stock* microservice requires isolation among different events being processed to ensure stock correctness while *Payment* can safely disregard

Table 4.4: Transaction Overview

Business Transaction	# Microservices Accessed	# Events
Customer Checkout	7	10
Update Delivery	4	$\lfloor 10 * 2 * AVG \rfloor$
Update Price	2	1
Delete Product	2	1

ordering semantics on processing payments.

The table 4.4 presents an overview of the number of microservices accessed and the number of events exchanged per transaction. We refer to *AVG* as the average number of items from each seller per order. As we see in Section 4.5, this is defined by the number of items per order and the seller distribution.

4.4 Data Management Criteria

In this section, we discuss the criteria that an implementation of *Online Marketplace* should meet. The criteria reflect key principles and challenges of data management in microservices. Some criteria have multiple levels, allowing the benchmark users to choose the most suitable one that fits their specific requirements. Furthermore, our explicit criteria specifications facilitate conducting fair comparisons between different systems on the same basis.

4.4.1 Communication and Data Access

Microservices are functionally partitioned applications [204]. Two issues arise with this model:
(i) Microservices interact with each other through asynchronous events to carry out functionalities across multiple microservices. For example, completing a cart requires a composition of functionalities across *Cart*, *Stock*, *Order*, *Payment*, and *Shipment* through the events they exchange.
(ii) State management is divided into two categories: (a) Direct data access: Each microservice can directly access its own data (encapsulation principle); and (b) Indirect data access: Data within each microservice can only be accessed externally through predefined interfaces (e.g., seller dashboard) or be notified about state changes through events (e.g., *Cart* is notified about

product updates). It should be noted that this criterion does not require data from different microservices to be stored in different databases.

4.4.2 All-or-nothing Atomicity

The business transactions must comply with all-or-nothing atomicity semantics. This criterion is crucial to guard against crashes or performance degradation leading to failures in the middle of a business transaction.

4.4.3 Caching or Replication

Section 4.3.2 describes the case of *Cart* subscribing to product updates with the goal of ensuring that checkouts do not contain outdated product prices. As the strategy of implementation varies system by system, either cache or replication can fulfill this requirement. In this sense, we prescribe three possible correctness semantics:

(i) Eventual. Updates (price update or product delete) are processed independently, disregarding the order that they are generated at the source, i.e. *Product*.

(ii) Causality at the object level. Updates on the same product are processed sequentially in accordance with the order in which they were performed at the source.

(iii) Causality across multiple objects. All updates made by the same seller must be applied to the cart in the same order they were applied at the source, achieving read-your-write consistency.

4.4.4 State Safety Properties

4.4.4.1 Inter-microservices Properties

. Refers to a property that cut across microservices.

S#1: Cross-microservice referential integrity. *Stock* always references an existing product in *Product*. We take inspiration from the CAP theorem [102] to define two levels of consistency:

(i) Available System. A deleted product will eventually not be allowed to be reserved anymore. The transaction is considered committed after *Product* responds to the `Delete Product` request.

At this level, if there is a network error or failure of the microservices, the delete message (EY) may not reach *Stock*. That requires the inconsistency to be detected and resolved at a later point.

(ii) Consistent System. `Delete Product` request is committed only when both Product and Stock have committed, therefore Product and Stock are always consistent with each other. At this level, a deleted product will not be available for future checkout attempts.

S#2: No duplicate checkouts. The cart of a customer session must not be checked out more than once. This can be safeguarded by having mutual exclusion in each cart. Another way to ensure this property is by assigning a customer session ID to each new customer session. The ID is included in the payload of each checkout request submitted to *Order*. Upon receiving it, *Order* ensures the same cart checkout does not lead to duplicate order processing.

4.4.4.2 Intra-microservice Properties

These include properties that can be enforced by relying on only local states. They can be achieved through appropriate isolation levels.

S#3: No overselling of items. In *Stock* microservice, both available and reserved quantities cannot fall below zero for every item. In addition, the reserved quantity must never be higher than the available quantity.

S#4: Maintenance of customer statistics. The system tracks successful and failed payments and deliveries through increments of numbers. An appropriate isolation level is required to ensure that the increment in *Customer* is not missed.

S#5: Linearized product updates. Updates on each individual product's price or version in the *Product* state must be linearized.

S#6: Concurrent Update Delivery transactions must execute in isolation. Since several events can be generated during a *Update Delivery* transaction, it is essential that concurrent *Update Delivery* transactions do not operate on the records of the same shipments and packages. The purpose is to avoid emitting duplicated `ShipmentNotification` and `DeliveryNotification` events.

S#7: Stock operations triggered by events must execute in isolation. Processing events in *Stock* requires either serializable isolation or exclusive locks on the items contained in the event payload.

4.4.5 Event Processing Properties

Event Order. Event processing order impacts *Order* and *Seller* dashboard. A microservice data platform can provide two levels of event ordering guarantee.
Unordered: `InvoiceIssued`, `PaymentConfirmed` and `ShipmentNotification` events are processed arbitrarily, possibly leading to violating the natural order processing workflow.
Causally Ordered: `InvoiceIssued` must precede payment events. The `Payment Confirmed` event must always precede shipment events. Similarly, a `Shipment Notification` with the status 'approved' and `DeliveryNotification` must always precede the corresponding `ShipmentNotification` with the status 'concluded'.
Consistent Snapshot for Continuous Queries. For the two concurrent queries of the seller dashboard to be consistent with each other, their results should reflect the same snapshot of the application state.
Event Delivery. The data platform can provide three levels of event delivery guarantees, namely at-most-once, at-least-once, and exactly-once delivery. The delivery guarantee has impacts on how to implement continuous queries and business transactions in *Online Marketplace* to achieve correctness. At-least-once and at-most-once delivery can make queries inaccurate. For example, while at-most-once delivery provides a lower bound on the profits in the *Seller Dashboard* without replaying lost messages, at-least-once can provide skewed results without accounting for duplicate events.

For transactions, at-least-once delivery requires microservices to be idempotent to account for possible duplicate event processing. On the other hand, a timeout mechanism must abort transactions if at-most-once or exactly-once delivery is used to prevent stalled transactions from blocking the other transactions.

4.4.6 Performance and Failure Isolation

The functional partitioning of the application allows microservices to operate independently, offering benefits from the isolation of resources assigned to each microservice [204, 133]. Implementations of *Online Marketplace* can achieve isolation in two tiers:

(i) Each microservice's code/logic is executed on different computational resources, achieving performance and fault isolation at the application tier.

This minimizes the impact of resource usage and failures between different microservices.

(ii) In the database tier, database operations do not interfere with each other in terms of performance and failure, achieved through isolated resource allocation to databases.

4.4.7 Audit Logging

Audit logging is a critical concern in applications because it allows developers to track application events, such as user activities (e.g., checking out a cart), and state changes [64]. These events recorded during application execution serve as an audit trail, helping developers troubleshoot faults and verify compliance with prescribed business rules. This is even more pressing in distributed systems, such as microservices. The substantial exchange of events and the complex interplay of independent components make it challenging to reason about errors and failures involving multiple asynchronous microservices. Thus, data systems must store audit events durably.

Apart from logging the events produced and consumed by each microservice, which is naturally achieved by message-oriented middleware with durability properties, in *Online Marketplace*, historical records of operations related to the events `ShipmentNotification` and `PaymentFailed` must also be logged to aid troubleshooting routines:

(a) Upon `ShipmentNotification` with `status` 'completed' or `PaymentFailed`, *Order* logs all records associated with such an order, in particular the relations *order*, *order_items*, and *order_history*.

(b) Upon `ShipmentNotification` with `status` 'completed', *Seller* logs all records associated with such shipment, in particular the relations *order_entry* and *order_entry_details*.

(c) As part of the emission of a `ShipmentNotification` with `status` 'completed', *Shipment* logs all records associated with such shipment, in particular those in the relations *shipment* and *package*.

(d) As part of payment processing, *Payment* logs payment records independently of the outcome (success or failure), particularly the relations *payments* and *payment_cards*.

4.5 Data and Workload Composition

The workload submitted to *Online Marketplace* can be adapted to fit particular needs. In the following, we present the different configuration parameters that can de defined for experiments.

Data Population. The state of some of the microservices require initialization prior to workload submission. The following procedure is expected, where X, Y, and Z are configuration parameters:
(i) X number of customers are inserted into *Customer* microservice;
(ii) Y number of products are inserted into *Product* microservice;
(iii) Y number of stock items are inserted into *Stock* microservice. Each *stock_item* tuple must refer to an existing product (through *seller_id* and *product_id* columns) in *Product* microservice;
(iv) Z number of products per seller leads to (Y/Z) seller tuples inserted into *Seller* microservice. In case of a positive remainder, the last seller must own W products, where $W < Z$.

On the one hand, the number of customers should be large enough to accommodate the maximum amount of concurrent transactions running the system at a given time. For instance, considering a throughput T, X must be higher than T to guarantee there is always a customer available. On the other hand, the number is limited by the amount of memory available.

Furthermore, the number of products and sellers should be large enough not to create many conflicts when running. Considering *prob_s* and *prob_p* as the probabilities of picking a seller and of picking a seller's product, respectively, to have less than δ conflicts, one should pick distributions that lead to $T * prob\_s * prob\_p < \delta$.

To aid data population, the dataset can be generated based on a given *size factor* following a uniform distribution. A size factor S leads to 10K customers (X), products, and also associated stock items (Y). Benchmark users can populate records in other microservices to simulate preexisting data and introduce a degree of overhead in state accesses during the workload execution. A size factor S leads the *order* table to be initialized with S*100K tuples uniformly distributed among customers, having the number of *order items* randomly picked (1-10). In consequence, *payment*, and *shipment* follow the same size given their tuples refer to an existing order.

Scale Factor. The customer checkout offers different parameters.
(i.) Each cart can have from 1 up to N items, chosen from a uniform distri-

bution. N is the *scale factor* that determines the size of a transaction and its execution cost.
(ii.) The quantity per cart item. A random value from 1 to a constant (e.g. 5), defined via a parameter, is selected.
(iii.) Probability of cart abandonment. A parameter is defined prior to workload submission, ranging from 0 to 100. A random number is selected and, if it falls below or is equal to the parameter defined, then the cart is abandoned before checkout submission.
(iv.) Probability of having a discount applied to a cart item. First, a uniform distribution is used to define whether an order will contain discounts. If the output number falls below or equal to the probability of having a discount, then a discount is selected next.

To define the discount to be applied, a random value from 1 to N is selected. N is the percentage of discount a customer gets for a product based on the observed price. Each discount leads to an additional record being recorded in *Payment* microservice state (relation *order_payment*) in case of a successful payment.
(v.) The payment method used. It governs how payment methods are selected. A uniform distribution is used to select either a bankslip, a credit card, or a debit card. A credit card payment method leads to an additional record being written to the payment's state (referring to relation *order_payment_card*).
(vi.) Probability of payment rejection. It governs the likelihood of a payment being rejected. A parameter is defined prior to workload submission, ranging from 0 to 100. A random number is selected and, if it falls below or is equal to the parameter defined, then the payment gets rejected.
Workload Distributions. Distribution in the workload is centered on sellers and their products. For every operation involving a product (add cart item, product price update, and product delete), one has to pick first a seller and then proceed to pick a corresponding item from the seller's product keyset. In this way, Uniform and Zipfian distributions can be used interchangeably in two cases:
- Seller selection: for every product selection, one has to first pick a seller based on a defined distribution.
- Product selection: After a seller is previously selected, one picks a seller's product based on a defined distribution.
Metrics. We collect two metrics in this benchmark. Throughput is the number of transactions processed per second, which includes both business

transactions that have been successfully completed and continuous queries that have successfully returned a result. End-to-end latency is measured from the moment a client sends a transaction request until the result is received. **Transaction ratios** define the probabilities of clients submitting various types of business transactions and continuous queries to the system. We envision it can be set to reflect some potential scenarios:

A. Order-heavy scenarios reflect e-commerce scenarios and high-sales periods, the workload is dominated by order processing (i.e., checkouts and deliveries), having a small percentage of abandoned carts and payments declined. This workload includes both uniform and non-uniform seller distributions, which models for example, some sellers' products being much more popular than others, leading to orders being canceled due to unavailability. The parameters are inspired by Olist workload [182] and specified as product delete with 2%, price update with 3%, seller dashboard with 5%, customer checkout with 30%, and update delivery with 60%.

B. Update-heavy scenarios reflect frequent updates with low-latency requirements, as found in banking, financial trading, and airline applications. This workload models a substantial degree of product updates with non-uniform seller and product distributions. Varying the zipfian constants allows for controlling the number of accesses to Cart and Stock, similar to some extended versions of Smallbank [223, 157]. One can set the zipfian constants ranging from 0.2 to 1.2, and transaction ratio as product delete and price update with 30% each, seller dashboard with 2.5%, customer checkout with 12.5%, and update delivery with 25%.

C. Hybrid transactional-analytical scenarios model a substantial number of users continuously querying their data (i.e., the seller dashboard) while the system must also cope with moderate transactional requests. The goal is to prevent the use of simplified caching mechanisms. Thus, seller distribution is uniform while products may have skewed access. Transaction ratios are inspired by HTAP benchmarks [51, 276] and defined as seller dashboard with 48%, customer checkout with 16%, update delivery with 32%, product delete and update price with 2% each.

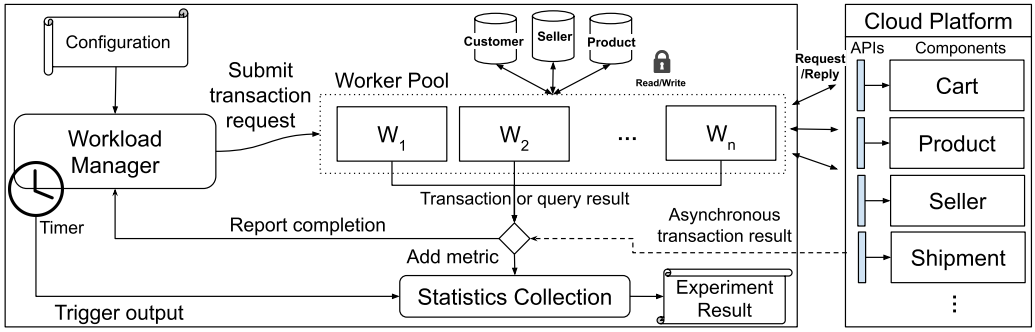


Figure 4.3: Driver Workload Manager

4.6 Benchmark Driver

4.6.1 Driver Functionalities

To manage the life-cycle of an experiment composed by: (i) data generation, (ii) data ingestion, (iii) data check, (iv) workload submission, (v) collection of results, (vi) report generation, and (vii) cleaning of states, we develop a benchmark driver in .NET [145].

A user specifies the data population, workload composition, ratio of transactions, and distributions through a configuration file. A user usually starts with generating data. Sellers, customers, products, and stock items are generated using Bogus [42], a popular library for generating synthetic but realistic data, including email, full name, address, and social security number. For products, Bogus generates matching product names and descriptions, including prices and locations. Generated data can be either stored durably or kept in main memory via DuckDB [72]. Users can load the data from DuckDB and move on to data ingestion, populating the microservices with initial data.

The driver's workload manager is exhibited in Figure 4.3. The workload manager ensures transactions are submitted within a time frame specified by the user. Besides, the driver controls the maximum number of concurrent transactions running in the target platform by (i) initializing a number of threads defined by the concurrency level. Each thread simulates a user interacting with the application; (ii) Whenever a transaction result returns, the driver pulls a previously used thread from a thread pool and spawns a new transaction submission.

4.6.2 Driver Implementation Challenges

In classical transactional benchmarks, such as TPC-C [55] and YCSB [53], the transaction inputs can be generated offline. However, we find offline generation unfit for the online marketplace benchmark.

Carts contain products, each with a price. The driver simulates customer and seller sessions. Customers add products to their carts, and sellers update their products' prices or even delete them. The former is particularly important to stress data replication and referential integrity properties. It is natural to deduce that if the cart is composed of products that significantly lag behind the latest updates to the application state, in the worst case, it can lead to not having any product to checkout or having all products with outdated prices. Both can deteriorate the workload distribution.

In case the cart formation and product updates were generated offline, to avoid the problem above, we would need to introduce substantial synchronization among worker threads (those that submit transaction inputs to the system), possibly leading to scalability problems in the driver itself. Therefore, this requires us to make the following advancements in *Online Marketplace* benchmark:

Simulating Customer Sessions. We start with the *Cart* microservice, which can be exemplified as a stateful operator. In the context of a customer session, a cart state evolves by having sequential operations (i.e., add cart item) up to a point where the state is sealed (through a checkout request), then returning to the initial state. To this end, the workload submission must account for each customer session individually. In case there is an active customer session for customer `X`, *there should be no concurrent driver's thread simulating the same customer*. The reason is that concurrent operations on `X`'s cart state can interleave, making it difficult to maintain the workload distribution.

To meet this requirement, a thread-safe queue for idle customers is maintained. Whenever a thread initiates a checkout transaction, it picks a customer from the idle queue to represent. After the checkout request is submitted, the thread pushes the customer to the queue, making it available again for other threads. The time a customer lies in the idle queue is driven by the concurrency level supported by the target platform. The faster checkouts are processed, the less time customers spend in the idle queue.

Managing Coherent Product Versions. As explained in Section 4.3.4.1, products could be deleted and replaced by new ones. In other words, new

product versions are being generated online continuously. It is crucial to generate transactions referring to coherent product versions. For example, to generate a price update transaction, we need to make sure to refer to the current product version in *Product*'s state. As another example, on adding an item to a cart, we need to make sure cart items refer either to the current product version in *Product* or the version that preceded the current one at the time of this operation. The goal is to simulate the latest product version "seen" by a customer.

To obtain a coherent product version while constructing transactions, we have to know the application state. However, the benchmark driver should not query microservice states during transaction submission for two reasons: (i) Workload generation should be independent of the actual data platforms being used; (ii) Querying the state of microservices would introduce additional load that is not prescribed in the benchmark.

In order to generate correct transactions while not querying the microservices' states, the driver manages a consistent mirror of the *Product* microservice state internally to guarantee access to coherent product versions. The driver linearizes the submission of concurrent update requests to the same product; the compare-and-swap mechanism is used to decrease synchronization costs.

Whereas the picking of product versions for building a cart item runs concurrently with product updates, so that customer and seller threads do not block each other.

Matching Transaction Requests to Asynchronous Results. It is common that platforms for building microservice applications provide results asynchronously (Section 4.6.1). In such an asynchronous system, the driver must track each submitted transaction and match it with the corresponding asynchronous transaction result to compute the metrics accurately. To this end, each transaction request is assigned a timestamp and a unique ID. This ID is later used to match the request to a corresponding transaction result that is eventually received. To avoid threads blocking each other, ID generation is decoupled from transaction submission. In conclusion, a stateful driver is necessary to provide correct transaction input.

4.7 Case Studies

To demonstrate the applicability of *Online Marketplace* and show the design implications and effects on different systems, we implement and experiment five versions of *Online Marketplace* in three competing platforms, Orleans, Statefun, and a custom solution. Orleans and Statefun are designed to develop event-driven services with state management functionalities. The custom solution is based on a combination of multiple systems that reflects an usual architecture seek in practice [150, 133], which will also provide an insight into the complexity faced by the developers. From the implementations and experimental study, we derive lessons learned (referenced by **L#**) and design decisions (**D#** in § 4.7.2.3) to foment new systems' design and improve both state-of-the-art platforms, highlighting the usefulness of *Online Marketplace*.

4.7.1 Implementations

4.7.1.1 Microsoft Orleans

Orleans is a framework for building distributed stateful applications. In Orleans, applications are composed of concurrent virtual actors [34], each encapsulating a private state, that are allocated transparently across machines [170]. Developers are offered APIs to log actor state [172], which allows actors to recover their state upon crashes or server migrations. In our implementation, we use the default Orleans actors' memory storage API to store the microservices data.

Due to the single-threaded actor abstraction, developers are encouraged to decompose application functionalities into distinct actors to avoid a few specific virtual actors becoming the bottleneck. Inspired by the guidelines of Wang et al. [265], our design aims to maximize parallelism and minimize transaction latency by assigning *Online Marketplace* functionalities to different actors. Details about the components' design is found in Table 4.5. We implement *Online Marketplace* on Orleans 7.2.1, using both the default non-transactional Orleans API and the transactional API, referred to Orleans Transactions (TX) in the rest of the paper. To allow Orleans actors to be reachable from the driver, we deploy an HTTP server on top of the Orleans silo. The server is responsible for parsing and forwarding incoming requests to the appropriate actors, reporting back the transaction results via

Table 4.5: Components Design

Actor/Function	Description
Cart	Models a customer's cart state and behavior
Customer	Models a customer's state and behavior
Seller	Models a seller's state and behavior
Product	Models a product's state and behavior
Stock	Models a stock item's state and behavior
Order	Models a unit of order processing for a single customer and associated state
Payment	Models a unit of payment processing for a single customer
Shipment	Models a unit of shipment processing for a disjoint group of customers and related state

an HTTP interface.

4.7.1.2 Flink Statefun

Statefun is a platform built on top of Flink for running distributed applications based on the concept of stateful functions [239]. Flink processes manage the state, handle incoming requests, and invoke the appropriate functions [237]. Each function encapsulates its own logical state and reacts to incoming messages asynchronously. Each incoming message is processed sequentially, in a way similar to Orleans actors. As a result, applications can be designed by composing functions through messages. Similarly to Orleans, we use the Statefun memory storage API to store the microservices data.

We implement *Online Marketplace* on Statefun 3.3, and, given the resemblance of both programming models, we opted to model stateful functions with the same design as Orleans (Table 4.5). To match the architectural design found in web services, we deploy Statefun with an HTTP ingress. Once the driver submits the transaction input, the ingress acknowledges the reception of the request and dispatches it to the appropriate function. We also deploy an HTTP egress to allow for the collection of transaction results that are eventually completed. The function that terminates a transaction sends a completion message to the egress. The egress operator stores the result and makes it available to clients. This design follows the Statefun

documentation [238]. To execute Stateful Functions runtime, we follow the recommended deployment mode using docker images and the most performant execution style (co-located functions) [236].

4.7.1.3 Implementation Insights

Upon implementing the *Online Marketplace* features in each platform, we encountered some limitations. We explain in the following how we mitigated them and the criteria that the target platforms can meet. Note that we only consider features that are natively supported by the framework, and use as few external systems as possible.

Data replication. Given that both platforms do not provide indexing for the actor or function states, we cannot efficiently query which carts contain a particular product. Although we could carry out this search iterating over all carts of the keyspace, we found this incurs a heavy cost operation, significantly impacting the benchmark analysis. Therefore, we opted not to implement this replication feature in the platforms.

Continuous queries. We do not use external stream processing systems in order to benchmark only the target platforms rather than the integration of multiple systems. Therefore, we opted to only implement the query dashboard, but not the continuous queries of cart abandonment and low stock warning. The reason is that the `order_entries` view can be maintained through conventional data structures and updated through application events, while the others require more complicated stream processing operators such as windowed aggregates and joins, which are not provided natively by the two platforms.

L#1. *Apart from being complex and error-prone, ad-hoc replication and continuous query implementations incur an overhead on both Orleans and Statefun platforms.*

Messaging delivery guarantees. Orleans provides an at-most-once delivery guarantee by default. Although Orleans can be configured to send retries upon timeout, we opted to capture the timeout exception and report to the driver that the transaction has been completed with an error. The reason is that, by enabling retries, the message may arrive multiple times, potentially corrupting data if the application is not idempotent [204].

On the other hand, Statefun manages state storage and message delivery in an integrated manner, such that in case of a delivery error, Statefun transparently retries the delivery up to a timeout and, upon that, rewinds

the application to a previously consistent checkpoint [78]. To make the performance results of the two platforms comparable, we disabled checkpointing in Statefun, and in case of a delivery error, we proceeded in the same way as in Orleans.

L#2. *The lack of exactly-once processing guarantees in Orleans requires developers to make their operations idempotent, adding execution overhead and increasing implementation complexity.*

Logging. We do not use Orleans storage because we run into the problem of inconsistent state while logging the actor's state [1]. Instead, we use external PostgreSQL to log completed transactions on both platforms. The consistent mechanisms also make the results of the two platforms comparable.

Resource Isolation. Both actors in Orleans and functions in Statefun share computational and data storage resources of the node they are allocated to. To isolate the computational resources of a specific microservice, it is necessary to configure a custom grain placement strategy in Orleans [192]. In Statefun, functions are allocated to Flink Task Managers transparently [85]. To make the results comparable, we use Orleans default placement strategy.

L#3. *The lack of explicit interfaces for configuring function allocation in Statefun can jeopardize achieving performance and failure isolation for microservices.*

4.7.1.4 A Composite Solution

Our experience above matches recent findings that microservice practitioners must combine several heterogeneous systems to fulfill their data management requirements [150]. Inspired by these results, we use a popular stack of technologies that practitioners often use to design a full-featured *Online Marketplace* implementation. The system architecture is exhibited in Figure 4.4.

To achieve all-or-nothing atomicity and concurrency control, we base our solution on Orleans Transactions. We offload consistent querying to PostgreSQL. With a combination of materialized view and refresh view triggered via actors, we achieve a consistent seller dashboard snapshot through a PostgreSQL transaction.

We implement two replication semantics: (i) Eventual, enabled by Orleans Streams [195], an Orleans add-on that allows actors to subscribe and publish to streams dynamically; and (ii) Causality across multiple objects, enabled by a primary-secondary Redis deployment. To decouple writers from readers, we use the following scheme: *Product* actors write to the primary

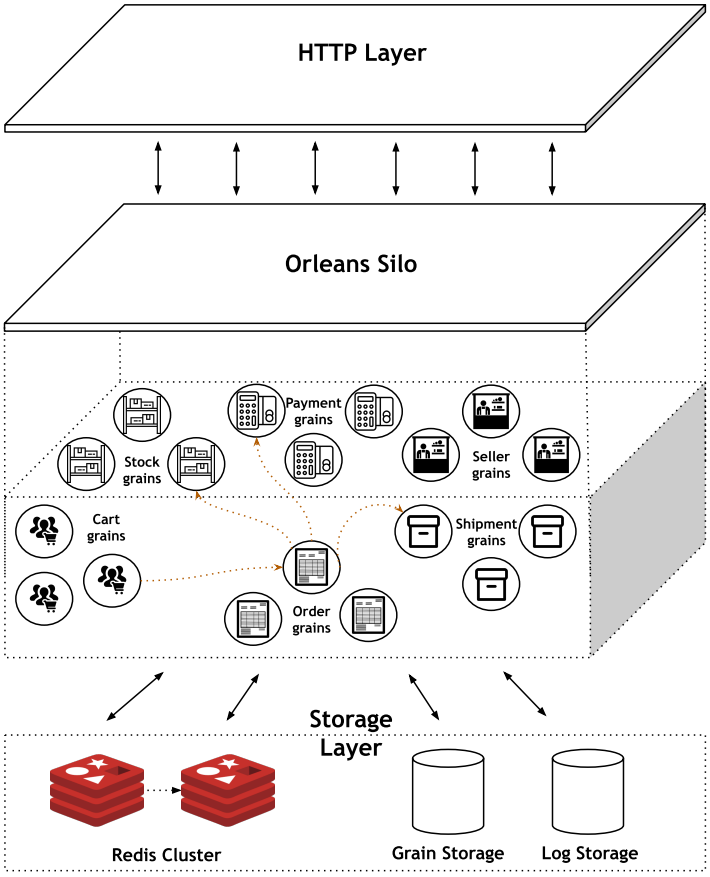


Figure 4.4: Customized Orleans-based Solution. *HTTP Layer* parses HTTP requests and forwards them to the correct grains. *Orleans Silo* provides location and life-cycle transparency for grains. Events are modeled as asynchronous messages exchanged by grains. *Storage Layer* contains a primary-secondary deployment based on Redis to support causal replication of product updates, grain storage to manage grain states and log storage to store audit logging.

node, and updates are asynchronously streamed to the secondary node, so *Cart* actors can read from it. As Redis linearizes writes, updates are applied to the *Cart* in order.

L#4. *Achieving consistent replication and querying requires weaving together heterogeneous systems through non-native APIs.*

4.7.2 Experimental Study

4.7.2.1 Experimental Settings

Deployment. We set up our benchmarking environment on UCloud [255] based on u1-standard instances. A u1-standard contains an Intel Xeon Gold 6130 CPU@2.10 GHz, 32 CPUs, and 384 GB of memory [254]. UCloud instances run inside a Kubernetes cluster connected via 100Gbps Infiniband virtual network. To avoid resource competition, we allocate independent instances to different benchmarking components, namely, the benchmark driver, the target platform, the PostgreSQL database server, and the Redis instances. To remove cache effects, we restart Orleans and Statefun after each run. In experiments involving Redis and PostgreSQL, we also clean up the state after each run. Besides, after data population, we ensure CPU usage returns to idle before initializing transaction submission. We use PostgreSQL 14.5 running in the operating system (OS) Debian 12.1, whereas the driver and platform instances run in the OS Ubuntu 22.10. All the instances are located in the same region and availability zone. The criteria in §4.4.6 are only applied to multi-node deployments. Since we focused our investigation on the performance and overhead of the platforms in multi-core settings, these are not applicable.

We followed Flink's configuration[1] and memory tuning [2] guidelines in order to configure and tune Statefun for our experiments. The configuration used is shown in Listing 4.4.

```
1 state.backend=hashmap
2 jobmanager.memory.process.size=24 gb
3 taskmanager.memory.process.size=24 gb
4 taskmanager.memory.managed.size=0 gb
5 taskmanager.numberOfTaskSlots=6
```

[1]https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/deployment/config

[2]https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/deployment/memory/mem_tuning

```
parallelism.default=6
statefun.async.max-per-task=16000
maxNumBatchRequests: 15
```

Listing 4.4: Statefun Configuration

Methodology. All experiments are run in 6 epochs of 10 seconds each with the first 2 epochs as a warm-up period. We collect two metrics: throughput and end-to-end latency. For each experiment, we maximized the resource allocated to the instances running the benchmark driver and PostgreSQL and we made sure resource usage was kept under 80%, to avoid them becoming the bottleneck.

Workload and Parameters. We use the workload modeling **scenario A** (§ 4.5). We set the probability of payments accepted and checkout to 100% to maximize the amount of transactions. Checkouts contain up to 10 items and quantities are selected randomly for each item (1-5). Each cart item has 5% probability of being assigned a discount. The discount value is randomly picked, not surpassing 10% of the item's price. We use both uniform and skewed distribution for picking sellers per each transaction. The number of sellers and products are picked reflecting *size factor* 10, so to not introduce substantial conflicts in uniform distribution.

4.7.2.2 Experiment Results

Driver Scalability. To examine if the coordination mechanisms of the driver (Section 4.6.2) are scalable, we conduct a scalability experiment. We use the parameters of **scenario A** (§ 4.5) and fix the transaction latency to 100 ms, varying the number of concurrent workers submitting transactions. We observe in Figure 4.5(a) that the number of completed transactions scales with the concurrency level, showing stable throughput across epochs. The overhead entailed by adding more worker threads does not create a bottleneck, independent of the distribution picked for sellers and products, reflecting the efficiency of the driver's coordination mechanisms.

Effect of Concurrency Level. In this experiment, we refer to *concurrency level* the maximum number of concurrent transactions running in the system at a given time. We measure the overhead incurred by different concurrency levels and identify the parameter that provides the optimal performance in each platform for further evaluation. In this section, we maximize the resources available (32 CPUs) and we set the workload skewness to be uniform

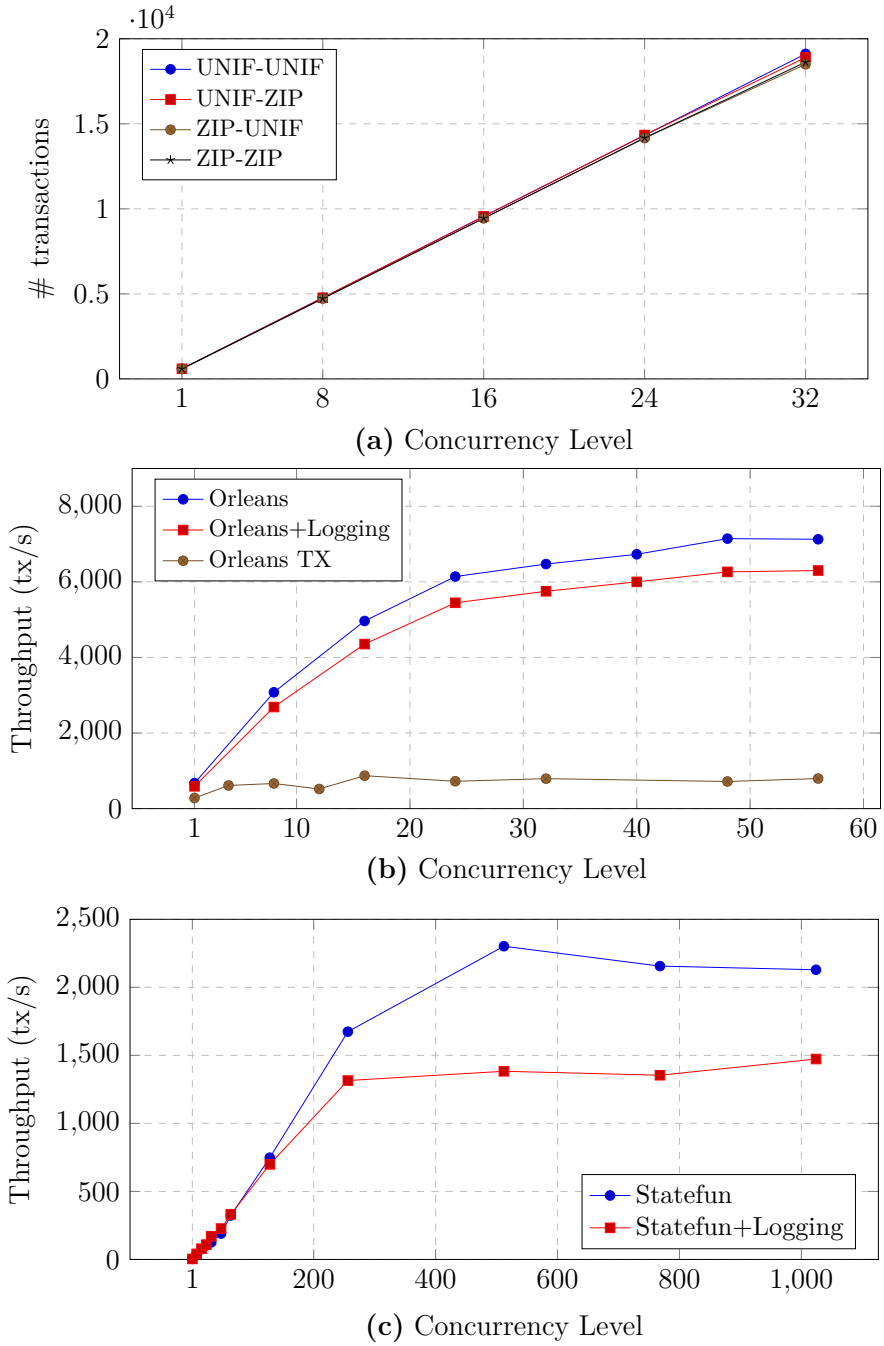


Figure 4.5: Driver Scalability (a) and Concurrency Overhead (b & c)

for both sellers and products.

As shown in Figure 4.5(b), Orleans achieves maximum throughput with the concurrency level 48. With an even higher concurrency level, more messages would be accumulated on actors' input queue, particularly actors in the checkout critical path (e.g., with *Order* and *Payment* doubling their processing latency). The overhead introduced by logging is negligible at the start and remains constant as the concurrency level increases, achieving a maximum of 12% overhead compared to non-logging. The lower throughput derives from the wait introduced by updates submitted to PostgreSQL.

As for Statefun (Figure 4.5(c)), in order to achieve maximum throughput, the concurrency level required is significantly higher compared to Orleans. We conjecture several reasons: (i) Although we aimed for the most performant deployment style, the containerized deployment mode introduces overhead on execution functions due to the virtualization layer; [3] (ii) The HTTP ingress acts as a single operator, processing requests serially, introducing a bottleneck; [4] (iii) Although we made sure to adjust the polling rate to a configuration that maximizes Statefun performance, the polling for transaction results coming from the driver to compute metrics in a timely manner invariably introduces a processing overhead.

Although negligible initially, logging shows a steep increase from the 200 concurrency level, maintaining a stable overhead afterward. The difference in overhead from Orleans lies in the benefits of reentrancy [194] found in virtual actors, which minimizes the effect of waiting for responses from external systems.

L#5. *The inherent deployment and processing model of the dataflow architecture of Statefun are key factors that harm performance in transactional and event-based workloads.*

Effect of Workload Skew. A skewed workload causes the access of certain records to become more frequent over time. Therefore, it increases contention on a small subset of actors/functions. While we measured the platforms under a low skew level in the previous experiments, in this one, we use the `zipfian` function API in the MathNet library [165] to generate different skewed workloads. To this end, we select different `zipfian` constants to vary

[3] We could not confirm this overhead because we do not find instructions in the documentation to run Statefun in bare metal.

[4] We tried using RichParallelSourceFunction to enable parallel ingress, but we found that driver requests were arriving out of order in functions, impeding the execution of some transactions.

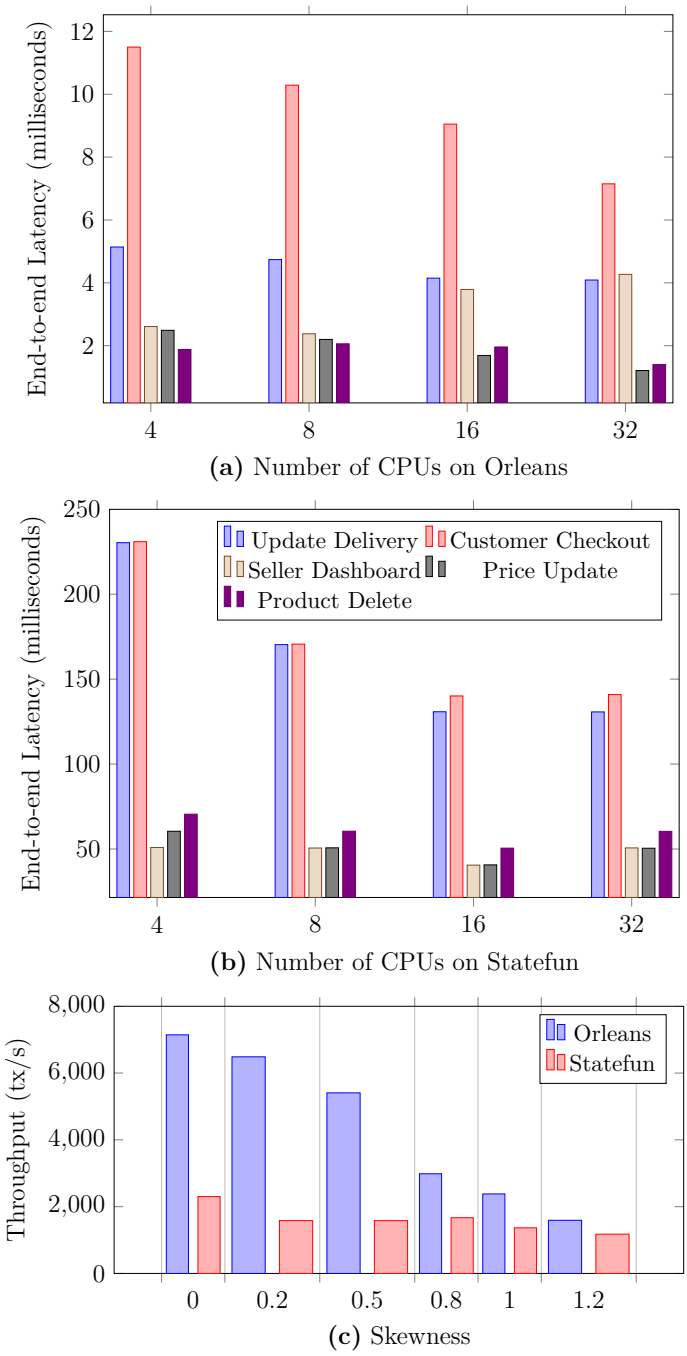


Figure 4.6: Breakdown Latency (a & b) and Workload Skewness (c)

the degree of contention in the workload.

Preliminary experiments showed that fixing seller distribution and varying product skewness showed similar throughput across different skew levels, which led us to investigate the effects of seller skewness. Thus, we picked the sellers using a Zipfian distribution while picking products using a uniform distribution. Again, we use 32 CPUs and the concurrency level that maximizes throughput. Figure 4.6(c) exhibits the Zipfian value of six skew levels used. [5]

It is observed that the throughput of Orleans decreases with increasing skewness, following a stable trend. This phenomenon is expected since there is a high contention on certain sellers and their products. Statefun, on the other hand, is less sensitive to workload skewness. In this case, we conjecture that the effects of batching messages play a role, allowing the scheduling of functions not to incur overhead as skewness increases.

L#6. *A skewed execution introduces a lower performance penalty in Statefun given the inherent batch-oriented execution model.*

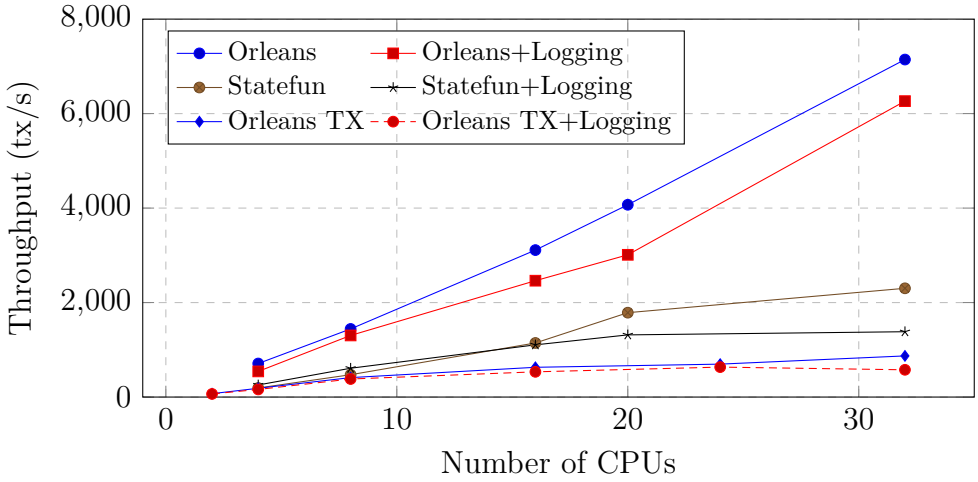
Scalability. In this experiment, we evaluate the scalability of both platforms by measuring performance metrics as more resources become available. We do so by increasing the number of cores from 4 to 32 and varying the concurrency level accordingly.

Figure 4.7(a) shows Orleans scales linearly as we increase the number of CPUs. In a similar way, Orleans with logging scales nearly linearly. The effects of logging remain low at the start, slightly increasing from 16 CPUs on and remaining stable afterward, matching the expected overhead found in concurrency experiments (Figure 4.5(b)). Statefun can also take advantage of increased computational resources but to a lower degree compared to Orleans. The overhead of logging is nonexistent at the start, and from 16 CPUs on, it impacts Statefun scalability.

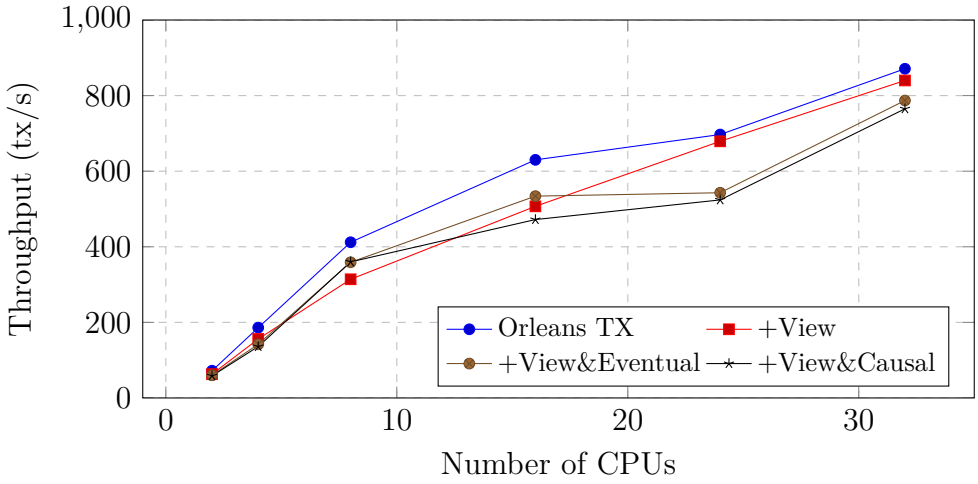
On the other hand, we found Orleans Transactions add a significant overhead by enforcing all-or-nothing atomicity and concurrency control. Even runs with a lower concurrency level are dominated by queued lock requests, often leading to message timeouts, thus affecting throughput significantly. We attempted to tune the timeouts of message and lock requests, but no configuration led to improvements over the default one.

L#7. *Ensuring all-or-nothing atomicity and concurrency control appear as dominant overheads on both platforms.*

[5] Statefun crashes on skew levels 1 and 1.2. The metrics are captured up to the crash.



(a) Data Platforms



(b) Composite Solution

Figure 4.7: Scalability

Figure 4.6 shows the latency breakdown of the transactions. In line with the throughput, customer checkout, and price update transactions, end-to-end latency in Orleans decreases as more CPUs are available. On the other hand, the seller dashboard shows increasing latency because the more CPUs available, the more concurrent transactions exist in the system, which introduces more customer orders, thus increasing the processing time to compute the query result. Update delivery transactions are not affected by the same phenomena because the processing is independent of the number of orders in progress in the system.

As for Statefun, for customer checkout and update delivery, it is possible to observe latency significantly decreases as more CPUs are available, showing improvements up to 61% from 4 to 16-32. On the other hand, the overall throughput results are reflected in the longer end-to-end latency. One of the aspects that drives Statefun's overall latency higher compared to Orleans is inherent to the programming model. Whenever an operation involves interacting with multiple functions, a requester function must send a message to each recipient function and wait for the eventual arrival of the responses. The function must keep track of the responses received through custom-made code, which necessarily involves storing responses in the function's state. On the other hand, in Orleans, multiple actors' calls are encapsulated through promises and do not involve state operations. This scenario occurs in both checkout and update delivery transactions to meet properties 1, 3, 6, and 7.
L#8. *The absence of native primitives for coordinating asynchronous operations across multiple functions significantly impacts Statefun's overall performance.*

Turning our attention to the composite solution, Figure 4.7(b) shows the overhead of consistent query (+View) and different replication semantics (Eventual and Causal). The results show that the overhead introduced by causal replication is not substantial compared to eventual, which can motivate users to overlook weak semantics in their deployments. On the other hand, the overhead of replication tends to increase as more transactions compete for resources. In the eventual case, we conjecture that, as Orleans Streams are also implemented as actors [193], they compete with transactional actors for scheduling turns. Similarly, in the causal case, the cost of the increasing number of network I/O operations tends to overcome the benefits of memory-resident data provided by Redis. We omit the results with logging because all run with an average overhead of 5 to 10%. It is worth noting that we managed to implement the missing criteria adding

less than 15% overhead on average, **demonstrating that meeting all the benchmark criteria is realistic**. However, the composite solution performance is bounded by Orleans Transactions.
L#9. *Offloading replication and continuous queries to external systems inevitably leads to increased network I/O overhead.*

4.7.2.3 Design Decisions

In this section, based on the observations and lessons learned identified through our case studies, we devise a set of design decisions to advance data platforms and foment new ones.

In the **feature** space, we posit the following to address **L#1**, **L#4**, and **L#9**:
D#1. *Enhancing microservice platforms with native, system-level support for data replication and querying microservice states is a promising feature.* This must be paired with performant system-level implementations. The inherent design of existing data platforms may limit the embracement of these missing features, leading to the next two design decisions.
D#2. *Integrating external systems and their features into microservice platforms is another potential candidate to relieve the burden on developers.* The case study showed it is realistic to rely on features provided by external systems. However, fulfilling missing requirements must not jeopardize performance.
D#3. *Integrating the features from external systems must be accompanied by optimizations to hide latencies associated with cross-system coordination to prevent substantial performance penalty.*

In terms of **performance**, lessons **L#5** and **L#8** demonstrate the values of the following design decisions:
D#4. *The event processing model and task scheduling in microservice platforms should reduce the impact of the long latency of certain multi-microservice workflows (e.g., customer checkout) on shorter transactions (e.g., price updates).* These facilities should ideally be transparent to developers, as in Orleans. In addition, designing primitives to circumvent concurrency limitations inherent to the programming model is another opportunity, as follows.
D#5. *Multiplexing multiple non-conflicting transactions within a scheduling unit (e.g., enabling grain reentrancy in Orleans) is a promising technique*

to speed up microservice workloads, enhancing throughput by running non-conflicting transactions concurrently.

From a **system architecture** perspective, lessons **L#2** and **L#6** derive the following:

D#6. *Holistically managing state updates and message queue operations found in Statefun appears to be a promising approach for ensuring exactly-once semantics.* That relieves developers from writing idempotent code to prevent duplicate executions, which may not always be possible. These should be paired with efficient solutions to schedule application functions.

D#7. *Batching multiple conflicting transactions is an effective performance technique for highly-contended microservice workloads.* Statefun inherits the operator execution model of Flink, which processes event streams in batches. Orleans, on the other hand, tracks message timeouts individually, which limits batching opportunities.

Lesson **L#7** derives better **algorithmic design** and lesson **L#3** addresses the usefulness of proper **abstractions**, respectively:

D#8. *Devising new algorithms and techniques for achieving all-or-nothing atomicity and concurrency control is critical for microservice performance.* Recent research shows it is possible to speed up Orleans Transactions by 4X through deterministic scheduling [157]. Offering weaker isolation levels that can still achieve the required properties is another promising direction. For instance, customer statistics can be updated in any order as long as the events are processed exactly once.

D#9. *Virtualized representation of application functions and associated states, such as through a virtual actor in Orleans, appears as an effective abstraction* to manage deployment, discovery, distribution, scheduling, and failures holistically in microservices.

4.8 Related Work

TPC-W [56] is a transactional benchmark that models the core aspects of user experience on an e-commerce website, such as browsing pages, checking out books, and searching for keywords (e.g., by title). TPC-C [55] was designed to represent the transaction processing requirements of a wholesale supplier. YCSB [53] models transactional workloads in the cloud that are not necessarily executed under ACID semantics. All of them assume a traditional monolithic application architecture. Our benchmark models a modern

microservice-oriented application and the complex interplay of their components through events, differing substantially from these benchmarks in terms of architectural style and data management requirements.

Unibench [277] offers OLTP and OLAP workloads for multi-model databases. However, it does not model the decomposition of microservices, which results in the absence of distributed and encapsulated states as found in microservice applications. On the other hand, stream processing benchmarks like Linear Road [7] focus on modeling continuous and historical queries, but they do not include transactional workloads.

4.9 Concluding Remarks

Our experiments with five implementations of *Online Marketplace* in three competing systems have shown varied results under different workloads, showing the ability of *Online Marketplace* to pinpoint missing core data management features and the impact of the programming abstractions and architectural design of these systems on performance. The experiments also highlight the overhead of all-or-nothing atomicity, concurrency control, constraint enforcement, replication semantics, consistent queries, audit logging, and asynchronous processing of events. **It is worth noting these cannot be obtained by existing microservice benchmarks (§ 4.2)**. Furthermore, from the lessons learned obtained through *Online Marketplace*, we derive design decisions to advance the development of data systems, highlighting that *Online Marketplace* will support designing futuristic data management systems for microservices.

Chapter 5

Transactional Cloud Applications: Status Quo, Challenges, and Opportunities

Transactional cloud applications such as payment, booking, reservation systems, and complex business workflows are currently being rewritten for deployment in the cloud. This migration to the cloud is happening mainly for reasons of cost and scalability. Over the years, application developers have used different migration approaches, such as microservice frameworks, actors, and stateful dataflow systems.

The migration to the cloud has brought back data management challenges traditionally handled by database management systems. Those challenges include ensuring state consistency, maintaining durability, and managing the application lifecycle. At the same time, the shift to a distributed computing infrastructure introduced new issues, such as message delivery, task scheduling, containerization, and (auto)scaling.

Although the data management community has made progress in developing analytical and transactional database systems, transactional cloud applications have received little attention in database research. This work aims to highlight recent trends in the area and discusses open research challenges for the data management community.

5.1 Introduction

In recent years, many applications such as Customer Relationship Management (CRM), reservation, and payment systems have been migrated to the cloud to take advantage of lower costs and elasticity. These applications were developed as monoliths, typically following the three-tier application architecture (presentation, application, data) [26]. In this architecture, business logic is implemented inside the application tier, while all data management takes place in the data tier, typically served by a database management system.

When migrating such applications to the cloud, developers need to split the functionality of a monolithic application to enable scalable deployment and development efficiency. This approach involves splitting monolithic applications into smaller and independent components that can be deployed and scaled separately, each serving requests as services. This design termed the *microservice* architecture, is widely adopted for migrating applications to the cloud. In the microservice architecture, each microservice is responsible for managing its own data (data encapsulation). Furthermore, implementing complex workflows spanning multiple microservices requires messaging and orchestration.

The emergence of cloud computing as a key paradigm for software and infrastructure as a service has prompted decision-makers and software teams to rethink their strategies for developing, deploying, maintaining, and modernizing their applications. In particular, researchers and industry are currently developing new programming models, data management methods, service communication models, as well as application deployment and lifecycle practices to exploit the low-access barrier to an unprecedented abundance of computing resources provided by the cloud.

To better align with the goal of on-demand, fine-grained resource provisioning enabled by the cloud and application performance requirements, modular software architectures, such as microservices, distributed application frameworks (e.g., Orleans [33], Akka Serverless) and the serverless computing paradigm, such as AWS Lambda [11], emerged as popular alternatives in the cloud application development landscape.

At the same time, microservices and application runtimes forgo key advantages that monolithic applications have relied on for decades: the delegation of state management, failure recovery, and consistency guarantees to

database management systems (DBMS). In modern microservice architectures, these responsibilities are intertwined with the application logic, mixing state management, messaging, and coordination into the application layer. From the perspective of the database community, the situation resembles the early days of computing, when developers relied on ad hoc, application-level transactions to maintain consistency [198].

In the last few years, the database community has focused on building and improving individual components used in cloud applications, such as serverless database systems and stateful functions. However, despite the pressing need for application migration to the cloud, the landscape of runtimes for transactional Cloud applications remains sparse. The programming paradigms and available systems differ substantially, each having key strengths and limitations; a characterization of challenges related to database research is still missing. In this work, we aim to explore the design challenges, clarify key differences between cloud programming paradigms, and highlight open problems and opportunities to evolve the landscape of cloud application runtimes.

Contributions and Outline. In this work, we propose a taxonomy (5.2) that centers around programming models, state management, and application lifecycle to tame the highly unstructured and heterogeneous cloud application landscape. The taxonomy reflects the building blocks that practitioners use and sets the stage to explore the state of practice for developing transactional applications in the cloud, along with their different designs and limitations. We then address open issues and research opportunities, highlighting how the database community can play a pivotal role in transforming the landscape of how cloud applications are built in the future.

5.2 Building Blocks

The programming abstractions offered by systems for developers developing transactional cloud applications are centered around three main building blocks, as shown in Figure 5.1: *i*) programming models (§ 5.2.1), with a focus on their parallelization primitives and how practitioners are building such applications; *ii*) messaging (§ 5.2.2) with focus on different ways of exchanging messages and performing remote procedure calls and finally; *iii*) state management (§ 5.2.3), with a focus on transactions and state consistency across services and scalability. Their interplay leads to trade-offs concerning pro-

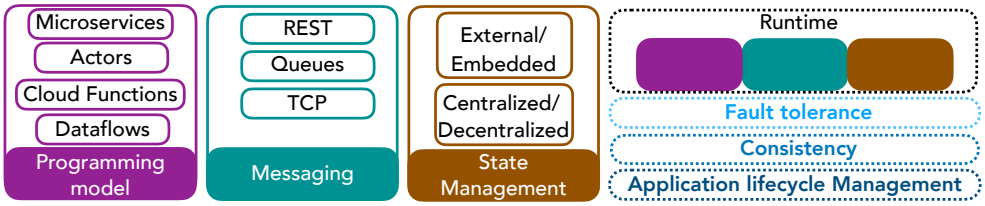


Figure 5.1: Building blocks and requirements for transactional cloud applications.

grammability, consistency, and performance, as recent findings suggest [150, 44].

5.2.1 Programming Cloud Applications

Programming models for distributed systems has been a long-standing line of research [22, 5, 274, 270, 21]. In the context of cloud applications, we identify that programming models play a crucial role in key system aspects, including but not limited to state and message management, fault tolerance, lifecycle management, and scalability. The status quo is the use of microservice frameworks (e.g., Java Spring [230], Python Flask [83]) and emerging programming models, namely Actors (e.g., Akka [4], Orleans [33]) and Stateful Functions (e.g., Flink Statefun [239], Azure Durable Functions [32]), all differing significantly in system model, abstractions, and guarantees offered to developers.

Therefore, we start with discussing the variables that drive developers to decide on a programming model, namely: (*i*) the programming paradigm, which relates to the application abstractions exposed to users (e.g., functions, actors, or objects); (*ii*) code modularity, which includes not only how program modules cooperate but also how application state is partitioned and encapsulated; and (*iii*) concurrency & transactional semantics.

Note that this work does not aim to provide in-depth analysis and formal semantics of models; rather, it focuses on how they fit in the cloud landscape and the main limitations of the systems enabling them.

Microservice Frameworks. To reap the benefits of parallel processing and loose coupling, the prevalent approach is functionally partitioning the application logic and state into independent components that communicate with each other via synchronous or asynchronous messages [204], called mi-

croservices. Microservice frameworks, such as Spring Boot (Java) [230], Flask (Python) [83], and Dapr (C#/.NET) [171], provide tools, libraries, and structures to help developers build microservices. These frameworks often include functionalities like Object-Relational Mapping for database interactions, service communication using REST or message queuing, and retrying features for fault tolerance. Each microservice built with such frameworks often employs a multi-threaded application server. Concurrency control and data consistency management are often provided by the underlying database system used by the component and the configured isolation level.

The Actor Model. The actor model is a programming model for concurrent and parallel computation in distributed systems [3]. An actor models a sequential process that performs transformations on the local state based on incoming messages. Actor systems are formed by a composition of actors, which communicate via asynchronous message-passing. Concurrency in actor systems is achieved by pipelining and dynamic creation of actors [3]. Traditional actor systems allow programmers to develop systems using low-level primitives by using actor IDs and prescribing their physical locations.

Virtual actors [33] are an extension of the traditional actor model that provides location transparency without forcing developers to deal with actor allocation in a cluster, life-cycle management, explicitly creating and tearing down actor instances, as well as failure transparency. It is currently found in popular distributed application frameworks like Orleans [33] and Dapr [171].

Cloud Functions. With the emergence of serverless computing [153], a new cloud paradigm called Function-as-a-Service (FaaS) [256, 32] rose in popularity. In FaaS, developers build applications as a collection of functions. Function executions are triggered by external events, such as clicks or invocations from other functions, allowing for function workflow compositions.

Initially, FaaS offerings targeted workloads with small to moderate I/O and communication, demotivating offering data models and consistency guarantees on operations within a single function or cutting across functions [279, 234]. More recently, though, there has been increasing interest in extending the FaaS paradigm to applications that access state intensively, called Stateful-FaaS (SFaaS) [279, 129, 234, 118, 206]. In SFaaS, developers also write programs based on composing functions and enjoy a key-value interface to access the global application state. Apart from the shared state interface, the programming, execution, and deployment model resembles Virtual Actors.

Stateful Dataflows. The dataflow model prescribes that an application is represented as a data flow graph. That involves decomposing programs into independent processing units. Organized as Directed Acyclic Graphs (DAGs), processing units (nodes) exchange data via message streams (edges). Dataflows have been mainly applied as the programming model for analytical batch and stream processing systems like Flink [37]. In these systems, processing units are framed as operators that can perform either stateful (e.g., joins, aggregates) or stateless (e.g., map, filter) operations. Message streams can be partitioned and assigned to different operator instances that execute concurrently. Stateful operators typically do not share state, preventing concurrency issues and enhancing parallelism.

However, the dataflow model has two main issues regarding its use for transactional cloud applications. First, dataflow systems are typically programmed using functional programming-style dataflow APIs, requiring developers to rewrite cloud applications to align with the event-driven dataflow model. While many cloud applications can be adapted to this paradigm, doing so demands significant programmer training and effort. Second, implementing *transactions* on top of dataflows, namely transactions that span multiple services with serializable guarantees, is still an open problem [239, 283, 65, 205].

5.2.2 Messaging

For any two components of a cloud application to communicate, a form of remote procedure call (RPC) is required, i.e., a way for a component to call a function on another remote component. In the past, RPC has taken multiple forms, such as Java's RMI [201] or CORBA's OMG [262]. Nowadays, most applications opt for either using REST APIs or message queues. We detail those below.

REST and gRPC. Built over HTTP (or HTTP/2), REST and gRPC are among the most popular ways to implement remote procedure calls for messages exchanging in microservice architectures. HTTP-based protocols [128] are typically stateless and cannot provide guarantees of message delivery. Thus, applications requiring message delivery guarantees must ensure these at the application level. Independently of the HTTP protocol adopted, a unique ID (e.g., in the form of an `idempotency key` [108]) is traditionally leveraged to prevent the execution of non-idempotent operations for incom-

ing duplicated messages. Messages are often duplicated in two cases: partial failures in the sender side and redelivery after a timeout. However, uniqueness ID guarantee and subsequent detection of duplicated messages are still the responsibility of applications, adding to the complexity of developing cloud applications [77].

Message Queues. Message queues (e.g., Apache Kafka [137], RabbitMQ [208], RedPanda [212], etc.) are typically used to implement asynchronous applications. The sender first pushes a message into a queue, the queue persists the message, and the receivers asynchronously pull messages from the queue to consume them. Producers and consumers are decoupled in time, facilitating the support to partial failures. On the other hand, receivers require acknowledging the consumption of messages. Despite the apparent simplicity, applications must coordinate the message processing and subsequent acknowledgment to prevent the execution of non-idempotent operations, a challenging task for many developers [150].

Relation of Messaging & State. As described above, an application state mutation depends causally on the arrival of a message. The operations over the state resulting from an incoming message must reflect in the receiver's state exactly once, characterizing the exactly-once processing guarantee. In sum, this means that the sender should be able to re-send messages to ensure the receiver has received them and, if a message is received multiple times, the receiver should be able to deduplicate them.

5.2.3 State Management

The state of an application usually refers to an application's data (e.g., the contents of a shopping cart or a bank account) that impacts its functionality and responses to client requests. Orthogonally to programming models and messaging, state management involves the placement and movement of data across components and strategies to make data durable and consistent. As depicted in Figure 5.1, state management in cloud applications depends on two main decisions: *i*) whether the state will be managed using an *embedded* approach (residing within the application runtime) or an *external* system, such as a database or a blob store, and *ii*) if the state access will be *centralized* or *decentralized*. In a centralized approach, the system manages the whole state in a unified way. In a decentralized approach, every subcomponent (e.g., individual operators in a stream processing system) handles its state

independently. In this section, we discuss the state management design space in cloud applications.

Microservices. There are two approaches to manage state in microservice architectures [150]: *i*) shared database, where data is logically separated (e.g., through private tables or distinct schemas), sharing database resources (i.e., centralized database); and *ii*) database per service (i.e., decentralized), where each service enjoys a dedicated database server, ensuring physical data isolation.

Physical isolation offers reduced coupling and independent scalability at the expense of higher complexity and infrastructure costs. On the other hand, a physically centralized database can impact teams by sharing database resources and artifacts (e.g., memory and disk resources, locks, or latches), jeopardizing performance isolation and application upgrades, respectively. Note that microservice architectures typically opt for the *external* approach to data management (§ 5.1).

Actors. Actor systems enforce logical state isolation, i.e., each actor manages and mutates its state. To this end, they often leverage language runtime and framework support to decouple actor calls from the actual execution (i.e., actor instantiation and memory address), inhibiting users from dealing with resource-sharing concerns [156]. Although actors typically keep their state private in main memory, some actor frameworks offer state management APIs that allow developers to store memory-resident states in durable storage [190]. Updates to an actor's state are only possible by messaging the actor. Depending on the actor system, mapping actors to servers can be both manual or dynamic [33]. Although it does not often affect how users operate over the actor state, actor placement can impact performance. In any case, data freshness guarantees are tied up to the most recent actor communication.

Cloud Functions. There are two dominant models for state management in the FaaS paradigm: private or shared state [234]. While in the former, the state of a function is modeled as an object that is tied to a given function, in the latter, functions are free to access any object, subject to the concurrency model imposed by the FaaS execution platform.

FaaS systems often ensure function invocations are scheduled in an individual computational resource, such as a container or a virtual machine [153]. In the first case, whenever a function is triggered, the corresponding state is brought from disaggregated storage to the memory of the compute nodes

assigned to run the respective function, all transparent to user code. In the second case, operations on shared state necessarily incur network round trips.

Dataflows. In most distributed dataflow systems, the application state is decentralized by design [37]. Typically, operators are scheduled for execution in separate nodes and rely on embedded LSM-based key-value stores like RocksDB [71] as a local state. Whenever the operator's state exceeds the local storage capacity, the state must be checkpointed, and the associated operator must be migrated to another node with sufficient storage capacity. Recently, there has been increasing interest in using tiered storage to battle scenarios where operators' states exceed local node storage [217, 168]. In this case, cloud object storage systems like S3 are used not only for checkpointing states [37] but also to store operators' states. It is worth noting that, unlike service-based architectures, state management in dataflow systems is transparent to developers.

5.2.4 Discussion

It is entirely possible to have a combination of programming models and state management primitives. For instance, Orleans can make use of an external database to store actor state, while there are dataflow engines that may store their state, instead of internally, to an external database system [109]. Although these approaches depart from the strict limits of the programming model or architecture at hand, they are valid deployment scenarios that are used in practice. In addition, low-latency microservices may need to embed a state to enhance data locality. Typically, a *cache* (e.g., Redis or Hazelcast IMDG) is used to speed up state retrieval, blurring the line between embedded and external state management. In any case, while mixing and matching different systems and approaches, deployments that go beyond the traditional settings also come with consequences in terms of fault tolerance and scalability, which will be discussed in the next section.

5.3 Requirements

In this section, we discuss the cross-cutting requirements of transactional cloud applications, namely: *i*) fault-tolerance (§ 5.3.1); *ii*) consistency (§ 5.3.2) and *iii*) application lifecycle management (§ 5.3.3).

5.3.1 Fault-tolerance

Microservices. Fault tolerance in microservices is achieved by making the application logic stateless and leaving state handling to an external database. Therefore, as long as a database of a given service is alive, the service operates normally. In case of failure at the stateless (application logic) microservices side, it is enough to restart a new service and connect to the same database. Although fault-tolerant by design, microservices may pose issues concerning state *consistency* due to the lack of a strong message delivery guarantee or transactional guarantee for multi-service workflows.

Actors. Modern actor systems have traditionally empowered three-tier architectures [33, 156], so developers checkpoint actor states to an external DBMS to ensure durability. As actor frameworks do not impose a database deployment model, ensuring performance, access, and failure isolation at the database tier is a non-trivial task at the hands of developers. On the other hand, actor frameworks like Orleans offer failure transparency by migrating actors across nodes in the presence of partial failures [33]. However, weak message delivery semantics and lack of transactional guarantees can leave actor states inconsistent after failures (§ 5.3.2).

Stateful Dataflows. For recovery, dataflow systems rely on checkpointing and logging mechanisms. Checkpoints in a distributed environment can be either independent per worker or in coordination by using a protocol [41]. Checkpointing ensures that the entire state is saved in (external) durable storage, and logging keeps track of all the data accesses between checkpoints. On failure, the system can retrieve its state by reloading the latest checkpoint, recalculating the state based on the actions saved in the log, and continuing from where it was left off.

5.3.2 Consistency

The consistency models in distributed systems reason about reads and writes on shared state and their real-time order guarantees across processes [248]. In cloud programming paradigms, though, we observe that the consistency models are inherently driven by the enabler systems' communication model and state management properties.

Microservices. Distributed applications designed through microservice architectures often remount the idea of the BASE model [204], characterized by

eventually consistent application partitions through queuing operations [114]. Practitioners also refer to this eventual consistency model through sagas [98] or patterns like orchestration and workflows [155].

Microservices often avoid distributed commit protocols to decouple components [150]. That would involve using language-specific libraries and implementing the protocol phases in each microservice, a complex and error-prone task for general developers. Besides, enabling the protocol across services is often impossible due to the lack of library support across heterogeneous programming languages and databases [135]. Most importantly, directly accessing data items in external services may break the desired state encapsulation, while the blocking nature of traditional protocol implementations affects performance.

Actors. With at-most-once messaging delivery guarantees by default, weak consistency across components is a popular design choice in actor-based applications. Some actor systems like Orleans allow customizable timeouts for retries to achieve at-least-once delivery. Statefun differs from Orleans in managing state updates and messages in an integrated manner, transparently rewinding the application state to a previously consistent checkpoint in case of a delivery error. Therefore, it achieves exactly-once processing and atomicity as a consequence. However, there is no transactional isolation across Statefun entities.

By default, Orleans provides no transactional isolation across actors. To enable transactional serializability in Orleans, users must utilize the Transactions API [191]. Apart necessitating porting the actor attributes to opaque objects [156], it has been shown to introduce a significant performance penalty according to recent experimental evaluations [157, 146], demotivating broader adoption.

Cloud Functions. Cloudburst enriches functions with causally consistent shared state accesses through a key-value abstraction [234]. Durable functions [31], in the context of Azure Durable Functions service, enhance FaaS with the ability to model entities (i.e., typed objects) as state abstractions for function manipulation, a richer state management abstraction than traditional FaaS offerings. Furthermore, individual function operations are atomic and enjoy exactly-once guarantees, guaranteeing atomicity in function compositions. Users must acquire and release locks explicitly to ensure transactional isolation on operations involving multiple entities (e.g., transfer money between account entities). However, there is no support for transactional iso-

lation across functions.

Another category of Cloud Function systems goes beyond by providing transactional serializability on computations cutting across functions [279, 129]. However, recent work [206] has found challenges in supporting large-scale, complex transactional applications like TPC-C in existing state-of-the-art SFaaS systems.

Stateful Dataflows. The dataflow programming model rose in popularity due to stream processing engines with support to exactly-once processing guarantees [37, 275]. Exactly-once guarantees eliminates the need for fault-tolerant code in the application since the engine transparently handles failures. However, exactly-once processing guarantees alone are not able to ensure transactional isolation.

5.3.3 Application Lifecycle Management

5.3.3.1 Resource Management

Resource management in cloud applications span a myriad of concerns, including the logical instantiation and localization of objects, failure handling, migration and deprecation of resources used by the application. The programming abstractions offered to developers (§ 5.2.1) also play a key role in application lifecycle management. We categorize lifecycle management into explicit and transparent.

Explicit. In microservice frameworks, application maintainers are in charge of deploying services, detecting failures, and implementing recovery routines. These implicitly include the objects managed by the application at run time, complexities that only exacerbate the existing challenges of maintaining consistent application states (§ 5.2.3).

Microservices are often deployed through resource virtualization mechanisms, such as virtual machines and containers. Although this scheme allows for isolation in case microservices share no (logical or physical) resources, as the resources are defined upfront, overprovisioning is a common challenge in service-oriented architectures. This is no different in managed runtimes such as Orleans and Statefun since application maintainers are responsible for provisioning and managing computational resources in a similar manner. Besides, actors and Statefun entities share computational resources, having developers limited interfaces to configure isolation. Furthermore, resource adaptation is also not transparent in microservice frameworks and managed

runtimes. Maintainers must explicitly adapt the resources available in reaction to higher performance demand, such as increased input rate, an often not optimal measure.

Transparent. The challenges above motivated the development of distributed systems and frameworks that transparently manage the life cycle of application objects. In frameworks that expose virtual actor abstraction, such as Orleans, users enjoy location and lifecycle transparency. Orleans allocates virtual actors on demand in healthy computational resources and deallocates resources used by actors when they are no longer used. In case of failures, Orleans transparently migrates virtual actors, ensuring that the application remains functional. However, users must handle resource provisioning and scaling explicitly.

Serverless runtimes, such as through FaaS systems, obviate the aforementioned challenges by offering transparent resource provisioning, function scheduling, failure handling, and elasticity to application maintainers. On the other hand, functions are ephemeral, having a bounded lifecycle (the start and end of a function), managed transparently by a FaaS runtime. Users trigger a function, often through a stateless request over the network, and are oblivious about how or to which computational resource the function will execute. FaaS systems ensure function invocations are scheduled in an individual computational resource, such as a container or a virtual machine. Whenever a function is triggered, its state is brought from storage to the compute node's memory assigned to run the respective function, all transparent to user code. However, challenges associated with cold starts, execution performance, and costs, undermine a wider adoption of the FaaS paradigm in application architectures [153].

5.3.3.2 Change Management

The evolution of applications is a key concern in the software engineering lifecycle, and it is no different in cloud applications [209, 273]. In a distributed environment, this includes, but is not limited to, the deployment, upgrading, and deprecation of components, as well as changes in the data and event schema. Surprisingly, support for application evolution in cloud applications is limited, and upgrades are often handled via ad-hoc approaches that rely on the expertise of application maintainers for correctness. We discuss next the application evolution space for microservices, actors, and dataflow systems.

In microservices, data schema changes can be performed without coordi-

nating microservices when the database-per-service pattern is adopted [150]. However, event schema changes require a some level of coordination among microservices. The reason is that events trigger functions in downstream components, necessitating matching the producer's and consumers' event schemas.

To alleviate this burden, data serialization systems like Avro [9] are used to transparently manage event schema changes over time, allowing producers and consumers to remain compatible. However, data serialization tools are oblivious to application state, which can possibly introduce breaking changes. As component tables are often populated using data contained in events [196], populating newly introduced fields is an example of a task that practitioners must treat explicitly [117]. Besides, if application state requires non-null fields, removing fields from the event schema leads to breaking changes.

Orleans provide grain versioning [69], a mechanism through which developers can annotate their grain classes indicating an new version. Introducing a new grain version does not require restarting an Orleans cluster. However, rolling out a new grain version require deploying a new silo and deprecating old silos that contain the old grains, a non-trivial task. Furthermore, if the new grain version is not backward compatible, calls to the new grain version are always forwarded to a compatible silo, which may limit scalability properties of an Orleans cluster. For last, there is no versioning support to stateless workers and streaming interfaces, impacting changes in scale-out scenarios and streaming workloads, respectively.

Upgrading a dataflow application or migrating it to another cluster often relies on the existence of a consistent snapshot [84], obtained via a checkpointing mechanism [41]. Operators, upon restart, must load their respective states from a snapshot and restart the event processing from the corresponding event stream offset. Besides, as dataflow applications form topologies, changes in the topology also undergo the same process. However, upgrading Fink applications is not an automatic process and must be carefully coordinated by application maintainers. Apart of being unable to revert writes to external systems, other challenges with the checkpoint-centric process appear, such as aged snapshots (requiring replaying a lot of events, a long-standing process) and big state checkpoints (overhead in streaming external storage to operators node).

5.4 Open Problems & Research Opportunities

In this section, we describe a set of open problems in programming models (§ 5.4.1), state and messaging (§ 5.4.2) and benchmarks (§ 5.4.3).

5.4.1 Programming Models & Systems

The variety of programming models available, along with the associated trade-offs in designing applications, such as data partitioning, access, and storage, concurrent application logic execution, fault handling, upgrade support, guarantees during crashes and network partitions, pose challenges to application developers in deciding for an ideal model. Another factor that only exacerbates these challenges is the proliferation of terms like *entities* or *objects* [239, 155, 79], *workflows* [171, 155], *durable* [155], *stateful* [239, 155, 79, 206], *reliable* [79], and *virtual* [33, 171], which are not consistent across systems since they express varied guarantees for applications.

Apart from the dataflow and actor models [3], and more recently Durable Functions [32], many programming models used in the cloud today are not formalized. The lack of formalizations and semantics of programming models hinders the ability to reason about a cloud application's desirable properties (such as safety, liveness, and consistency), a key impediment to advancing cloud programming. Another direction that can mitigate some of these concerns is via declarative programming. Ongoing work in this realm includes stateful entities [207], HydroLogic [45], and event-based constraints [151].

Furthermore, systems should be designed to enable developers to effectively perform traditional software engineering activities. Limitations with debugging, application evolution, and observability are additional factors that demotivate the use of systems for cloud programming. An open question is whether it is possible to devise a programming model and system with transparent parallelization, scalability, and consistency.

5.4.2 State & Messaging

Data Model. Programming models used in the cloud often provide opaque state management abstractions [156, 33, 207, 114]. To fully realize the benefits of serverless computing, it is key that programming abstractions for the

cloud offer not only formal semantics but evolve to allow users to operate with richer data models and express cross-component data invariants [45] that are popular in practice [146], not jeopardizing state encapsulation.

Disaggregation. In the same line of industry-strength cloud-native database [259] and stream processing systems [100, 168], disaggregated storage [264] can be leveraged in cloud programming abstractions to support ever-growing data that applications process in the cloud. That must account for performance, access, and failure isolation properties, abstracting away these concerns from application code. Although most FaaS architectures provide disaggregated storage by design [153], challenges inherent to composing applications via ephemeral functions and shared state access, hindering programmability and encapsulation of states of cloud applications, respectively, are still present.

Most microservice architectures adopt persistent and asynchronous communication via message queue systems [248], with the message layer being disaggregated by design. However, considering most cloud applications rely on language runtimes such as Java's JVM and .NET's CLR, optimizing runtimes for cloud applications could focus on optimizing message transport, processing, storage, and recovery, considering the operating system and application interplay.

Consistency. Traditional approaches such as SAGAs [98] and OpenXA [228] allow for coordinating consistency guarantees across microservices. More recent work [82] introduces causal consistency for microservice architectures. Cross-engine transactions [282] is a promising approach since it operates at a lower level than the application. However, implementations should avoid exposing private encapsulated data and protocol details in application code. Coordinating with external, often legacy, systems is very common in cloud applications that developers currently handle in an ad-hoc fashion.

5.4.3 Workloads & Benchmarks

Benchmarking a distributed cloud application for performance and even correctness is largely a task that takes place in an ad-hoc fashion at the moment. Efforts such as DeathStar[97] have been used to evaluate distributed cloud application frameworks [279, 129] alongside TPC-C [206]. In addition, despite recent efforts to benchmark cloud applications [97, 286, 202], most benchmarks are oblivious to key aspects of data management. At the same time, traditional metrics such as throughput and latency used to benchmark

OLTP and OLAP systems may not suffice emerging cloud programming systems alone. Modeling request arrivals should consider systems' design goals and the cloud serving model used [219].

The use of event streams as a paradigm to compose applications and the presence of data invariants, transactional guarantees, data replication, and querying in real-world applications are examples of missing requirements for existing benchmarks. Recent work [146] aims to fill these gaps, but challenges related to dynamic workloads, observability, and recovery remain open.

Chapter 6

A Distributed Database System for Event-based Microservices

Microservice architectures are an emerging industrial approach to build large scale and event-based systems. In this architectural style, an application is functionally partitioned into several small and autonomous building blocks, so-called microservices, communicating and exchanging data with each other via events. By pursuing a model where fault isolation is enforced at microservice level, each microservice manages their own database, thus database systems are not shared across microservices. Developers end up encoding substantial data management logic in the application tier and encountering a series of challenges on enforcing data integrity and maintaining data consistency across microservices. In this vision paper, we argue that there is a need to rethink how database systems can better support microservices and relieve the burden of handling complex data management tasks faced by programmers. We envision the design and research opportunities for a novel distributed database management system targeted at event-driven microservices.

6.1 Introduction

Modern business scenarios require systems to make decisions in real-time based on events. To tackle such scenarios, event-driven architectures (EDAs) are often advocated as a compelling approach to meet the stringent requirements required by data-intensive systems that react to events [265]. An

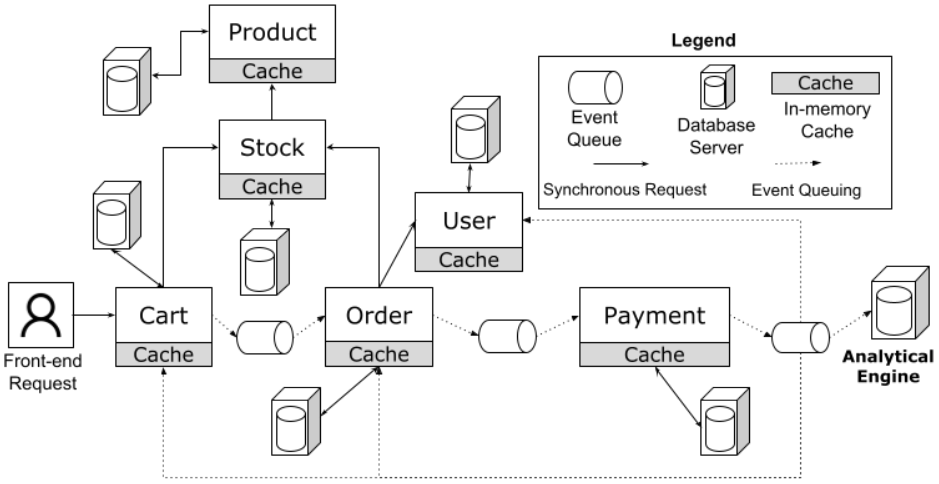


Figure 6.1: E-commerce microservice architecture example

EDA is composed by highly decoupled, single-purpose event processing components that asynchronously receive and process events, each performing a singular task in the application [213]. A particular approach to realizing an EDA is through the *microservice architectural style* [143], an emerging industry paradigm to design highly-modular and scalable applications [288]. In microservices, an application is designed as a set of small and independent building blocks that communicate with each other via pre-defined interfaces (e.g., HTTP APIs) or events. Each microservice may manage its own database, and thus select its underlying technology to best support the data formats and workloads for the computations encoded. Thus, microservices follow a *decentralized data management* principle. In line with this principle, the dominating practice is that microservices and their underlying databases are deployed in separated containers to preclude errors from propagating across microservice boundaries.

Motivating Example. Microservice applications deviate from traditional monolithic transaction processing, as illustrated by the e-commerce application shown in Figure 6.1. After adding several items to a cart, where a cart is an entity managed by the *cart* microservice, the customer may initiate the order's payment process. From here, different options could apply, e.g.: (i) for each cart item added, the *cart* microservice acquires a temporal (i.e., expiring) lock, a promise from the *stock* microservice that the stock item is safeguarded from being acquired by other clients; (ii) the cart items are

added arbitrarily (i.e., without any synchronization) and the *order* microservice is responsible for checking against the *stock* microservice the availability of cart items by the time a checkout request is issued by the user.

In either case, the *stock* microservice may verify whether the products are available and whether mismatches exist (e.g., in the product price). After confirming that all items are in stock, the order is then confirmed. However, prior to proceeding with payment, the *order* microservice retrieves user information from the *user* microservice and checks the validity of the user's discounts (e.g., in case of no longer valid discounts applied previously). After applying all the proper updates (e.g., calculating totals) to the customer order, the *order* microservice emits an event so that the *payment* microservice can proceed with the payment confirmation.

The *payment* microservice processes the payment by acquiring the confirmation of funds from the credit card holder (e.g., external system, often a blocking operation), and afterwards updates the user's credit score by contacting the *user* microservice. Lastly, an analytical engine is eventually updated with the new order.

Despite the apparent attractiveness of a loosely-coupled design and an inherently decentralized data management architecture, the presented application unavoidably requires substantial coordination across microservices, thus forcing developers to deal with challenges that would not arise in a traditional monolithic architecture. In particular, by following a model reminiscent of BASE [204] and OLEP [133], microservice developers end up encountering challenges that should have been solved by database systems. For instance, these models prescribe decomposing a schema and associated application logic into functional partitions, hard-wiring functional dependencies through asynchronous events. As result, these models force a substantial amount of data management tasks into the application level, such as enforcing constraints cutting across several microservices (e.g., referential constraints) and ensuring liveness, since a transaction is often broken down into several steps due to the functional decomposition as in our example. Furthermore, microservices introduce challenges not originally envisioned by the BASE and OLEP models, which we discuss next in the context of our example application.

Cross-microservice synchronization and validations. The substantial data management logic encoded in the application-tier creates a barrier for enforcing application safety across microservices. Developers are offered neither efficient nor intuitive interfaces [114] for encoding distributed syn-

chronization (e.g., locks and leases) at the application level correctly. As a result, microservice developers often end up resorting to eschewing synchronization altogether, leading to data consistency issues. For instance, in *eShopContainers* [76], the application we based our example scenario on, an asynchronous event generated by the *payment* microservice (after payment has been processed) triggers the removal of items from stock. However, the application code does so unsafely, i.e., races and failures can lead to inconsistent state being recorded across microservices.

Event-based constraint enforcement. Built from our example in Figure 6.1, consider the case where a *product-price-update* event is emitted concurrently with a *checkout-cart* event request. The order in which the *cart* microservice should process these concurrent events must depend on an ordering constraint, otherwise any arbitrary order may apply. However, programmers have no way to specify event processing order invariants related to the possible interleaving of event streams and end up encountering challenges to guarantee consistency.

Cross-microservice queries. Queries spanning multiple microservices are popular in microservice architectures [74]. To implement such queries, programmers often encode data processing logic for aggregating and joining data from different microservices at the application level. By resorting to such ad-hoc mechanisms, programmers are exposed to a myriad of anomalies, such as fractured reads [19]. Besides, faulty microservices are another challenge when retrieving data from multiple microservices. Here, the impossibility of accessing a microservice's private state may lead to an incomplete view of the state.

For instance, based on our example, suppose an analyst necessitates a report aggregating the last day's worth of historical data from users, their orders, and discounts. The lack of a principled approach for state management across microservices forces developers to resort to ad-hoc and error-prone mechanisms to query and join such data.

Data replication. Data replication is usually employed to alleviate the amount of requests required in queries spanning multiple microservices. Through propagated events, the OLEP model predicates that microservices can incrementally maintain materialized views composed by data owned by other microservices. However, given the heterogeneity of microservice databases, it is often the case that practitioners resort to *ad-hoc* application-level mechanisms to support data replication through events, a complex and error-prone approach that leads developers to make do with only eventually consistent

views. In our example, suppose the *Order* microservice maintains a materialized view of users' discounts asynchronously updated via events. Weak replication semantics may lead to issuing an incorrect discount to a user.
Fault tolerance. Consider the case where a business transaction performs writes to a microservice's private state and queues an event to a message broker (to trigger an operation in another microservice, as observed in the *order* microservice). In these cases, the BASE model advocates that all writes must be performed in the same resource (i.e., data store) [204] to avoid a distributed commit protocol. However, it is often the case that such support is not available, thus leading developers to resort to error-prone fault-tolerance strategies at the application level [8].
Contributions and Outline. The paper is organized as follows. Section 2 discusses the limitations of state-of-the-art database systems in light of the data management challenges described. Section 3 presents our contributions: (i) We argue that although traditionally applications have been treated as black boxes, such a paradigm is insufficient for addressing the data management challenges in microservices, as it offers no way to expose the complex data interplay outside the database among microservices. We thus advocate for identifying such complex interactions and data management tasks taking place at the application level and then pushing them down for database processing; (ii) We propose a novel abstraction called *virtual microservices*, to represent the computations performed by microservices inside the database; (iii) We present the declarative constructs to allow for the identification of virtual microservices at the application level, and; (iv) We present a vision for a microservice-oriented event-driven database system. Section 4 discusses challenges and research avenues for this novel database approach, and Section 5 concludes the paper.

6.2 Status Quo and Limitations

6.2.1 State-of-the-art database systems

Although it is possible to observe a variety of architectural designs in classic databases, separation of database applications is enforced at schema-level. This can be considered a weak functional isolation scheme since the performance degradation of a application may impact other applications. Besides, whenever distinct applications need to share data, it is assumed to take place

within the database tier, which does not meet the microservices' state of the practice.

Cloud-native multi-tenant databases [58] logically isolate tenants and provide elastic resources backed by the cloud infrastructure. However, they fail to support event-based programming and advanced data management requirements, such as cross-tenant data replication and computations. By assuming tenants are completely isolated, these systems cannot capture dependencies and interactions amongst microservices.

Taking a step back, there has been a tension yet not properly addressed by the database systems community between the needs and requirements of developers in the wild and the classic database abstractions, which treat the application side as a black box. This view clashes with the needs of programmers that prefer encoding their complex business logic in the application tier, which often relies on application-level feral validations [18]. This tension is worsened by the emergence of distributed applications that take advantage of the flexibility and cost-effectiveness offered by the cloud. In this case, computations are no longer being held based on a single database, but rather traverse several small building blocks of the application, often making use of a myriad of data systems, such as caching or pub/sub systems, to meet data management challenges. A holistic solution incorporating selected data management functions of multiple such building blocks is required to fully address the needs of microservice applications.

6.2.2 Stream processing systems

Usually framed as a compelling abstraction for microservices [131, 245], stream processing systems (a.k.a. dataflow systems) are designed to perform continuous queries over unbounded data streams [131]. The computational model of stream processing engines contrasts with microservices, since microservices are often independently developed by different teams and deployed separately, instead of within a single streaming processing engine, to provide strong isolation. At the same time, microservices are free to communicate, often through non-blocking primitives, and operate over data from other microservices. Besides, failures or changes in a microservice should not propagate over or interrupt other building blocks of the system, which contrasts with existing stream engines.

It has been recently argued that microservice applications can be built on streaming dataflow systems by making stream processors full-fledged data

management engines (e.g., by supporting for transactions across microservices) [131]. However, it remains an open question how to match static dataflow graphs prescribed by such a solution with the loose-coupleness, autonomy, and dynamicity principles of microservices [288].

6.2.3 Function as a Service

Although recent advances in serverless computing through the function as a service (FaaS) API [116] aim at offering an easy-to-use platform that provides programmers high-level computation expressibility and automatic resource management, FaaS is usually perceived as a fit for more stateless and less stringent data-intensive computations.

By contrast, one may position stateful functions as a proper abstraction for microservices, since they provide an API for state management that is particular to the business logic encapsulated by the function [245]. However, it remains an open question how stateful functions could support advanced data management features required in microservices, e.g., ordering constraints in complex interleaving of data streams, online queries, and cross-microservice synchronization and validations.

6.2.4 Frameworks for distributed applications

Orleans. While Orleans can be used to develop stateful middle-tier applications like microservices, for not being a full-fledged database system, it still forces developers to reason about application safety at the application level, such as the impact of the interleaving of events to private state and explicitly handling data durability concerns. Furthermore, applications must respect a strict set of characteristics to benefit from the Orleans paradigm [190]. In this sense, it is unclear how microservices, such as the example scenario, can be modeled through virtual actors [265].

Dapr. Dapr is a framework for facilitating the development of microservice applications [171]. By exposing a standard API for microservices to connect to the Dapr middleware, Dapr is able to intermediate message queuing across microservices. However, practitioners are still forced to deal with the aforementioned challenges at the application level. Most importantly, Dapr offers a centralized and homogeneous (i.e., key-value) state management abstraction, contrasting with the prescribed data sovereignty of microservices.

6.3 Microservice-oriented Databases

In this section, we describe our vision for a database architecture that targets providing a principled design to tackle the limitations found in state of the art database systems regarding meeting the data management challenges found in microservice architectures.

6.3.1 The gist

Although existing data systems partially support real-world microservice deployments, advanced data management features required by microservices are not explicitly or sufficiently addressed in conjunction. Particularly, application semantics are mostly unknown to the database, which hinders the database from being able to capture data management tasks encoded at the application level.

Given this clear impedance, we advocate for rethinking how database systems interact with this growing class of applications. We hypothesize that through appropriate abstractions, we can proactively identify and push data management functionality down to the database and provide built-in advanced data management support directly to the application.

Centralization vs. Decentralization. In order to achieve the necessary isolation between microservices, the conventional wisdom advocates a decentralized data governance paradigm, which is often implemented by using the database-per-service pattern. This paradigm is the root cause to the aforementioned data management challenges of microservices. To enable pushing down data management tasks to the database system, in contrast to the conventional wisdom, we propose a central data governance paradigm for microservices, i.e. using a single scalable and distributed database system to manage the states of all microservices. We argue that such a paradigm does not necessarily contradict the decentralized data management principle of microservices, as long as the developers are able to express the logical boundaries of each microservice, and the database system can consistently enforce such boundaries. As demonstrated by the success of multi-tenant database systems, which offer data management services to independent and isolated tenants through a central database, we believe that providing fault-isolation, performance isolation, and data sovereignty guarantees to microservices is independent from the database being centralized or decentralized. Further-

more, given the recent developments of HTAP and Polystore database systems, which are able to respectively cope with heterogeneous workloads and data formats, we observe that a centralized data governance paradigm can be made orthogonal from the database system being able to manage multiple underlying databases.

Key takeaway. Appropriate abstractions allow us to enrich knowledge of the data management logic carried out by the application and the complex interplay of microservices. This leads to proper division of tasks between the application and the database system, where data management tasks are pushed to the database, alleviating the burden on developers.

6.3.2 Virtual microservices

In order to push down data management tasks to the database system, a proper abstraction needs to faithfully characterize the semantics of microservice applications inside the database. This tension leads us to the following question: **How can we map a microservice application's invariants and data management tasks to an internal representation that the database can effectively manage?**

To address this question, we envision the notion of virtualized microservices as the core building blocks of a database architecture. In other words, each microservice application ought to have an abstract representation in the database, a *virtual microservice* twin.

More precisely, a **virtual microservice** is a construct that logically encapsulates the state of a particular microservice, along with its constraints and data dependencies, as well as abstracts inbound and outbound event streams. In other words, by exposing internal representations of microservices, the database system gains knowledge about the event dataflow across microservices in addition to the constraints that cut across microservices. As a result, *the application is no longer a black box to the database.*

As an idealized microservice twin, a virtual microservice is not only an independent entity, but also an internal representation of the application managed by the database. Thus, it inherits the same characteristics of a microservice, which must then be enforced consistently in the database, namely: (i) communication by event-based asynchronous messages; (ii) private mutable state; (iii) shared data that is not mutable. The objective is to natively support the data management tasks encoded in microservices within the

database, but at the same time provide features that normal microservices currently do not have, e.g., explicit data dependencies, integrity constraints, and atomic actions across private state and event streams.

Key abstraction. A virtual microservice is a database abstraction that reflects the inner characteristics of a microservice running outside the database, including inbound and outbound events, private state, foreign data dependencies, and integrity constraints.

6.3.3 A cross-stack architectural vision

The advanced data management requirements (§ 6.1) and shortcomings found in state-of-the-art database systems (§ 6.2.1) pose significant challenges to effectively supporting data-intensive microservices. It is unclear how a database system can jointly support event-based querying APIs, cross-microservice queries, event-based constraint enforcement, high consistency in data replication across microservices, and proper cross-microservice synchronization, while at the same time providing isolation boundaries between microservices.

To address this conundrum, we now turn our attention to our **success criteria**: *enabling pushing data management tasks to the database and materializing the virtual microservice abstraction into a principled database architecture that tackles the challenges of data management in microservices by design.* Our vision is shown in Figure 6.2, which depicts an event-driven microservice-oriented database architecture. Broadly, the architecture provides an application framework, which controls the interaction of the application tier with the system, and a set of well-defined components that target particular data management concerns and can scale independently.

We explain in the following the interconnection between the components, how the key insight and abstraction are enabled, and how the pressing challenges are met.

Application abstraction. To enable pushing data to the database, it is necessary to enrich the abstraction provided to the application to allow for identifying the complex data management tasks encoded. A simple but powerful tool that has been historically used to add functionalities seamlessly to applications is a framework [88]. Frameworks provide a dynamic introduction of behavior in the application without the need for user-defined code.

A framework is a key enabler not only because it is responsible for identifying data management logic and informing the database about the appli-

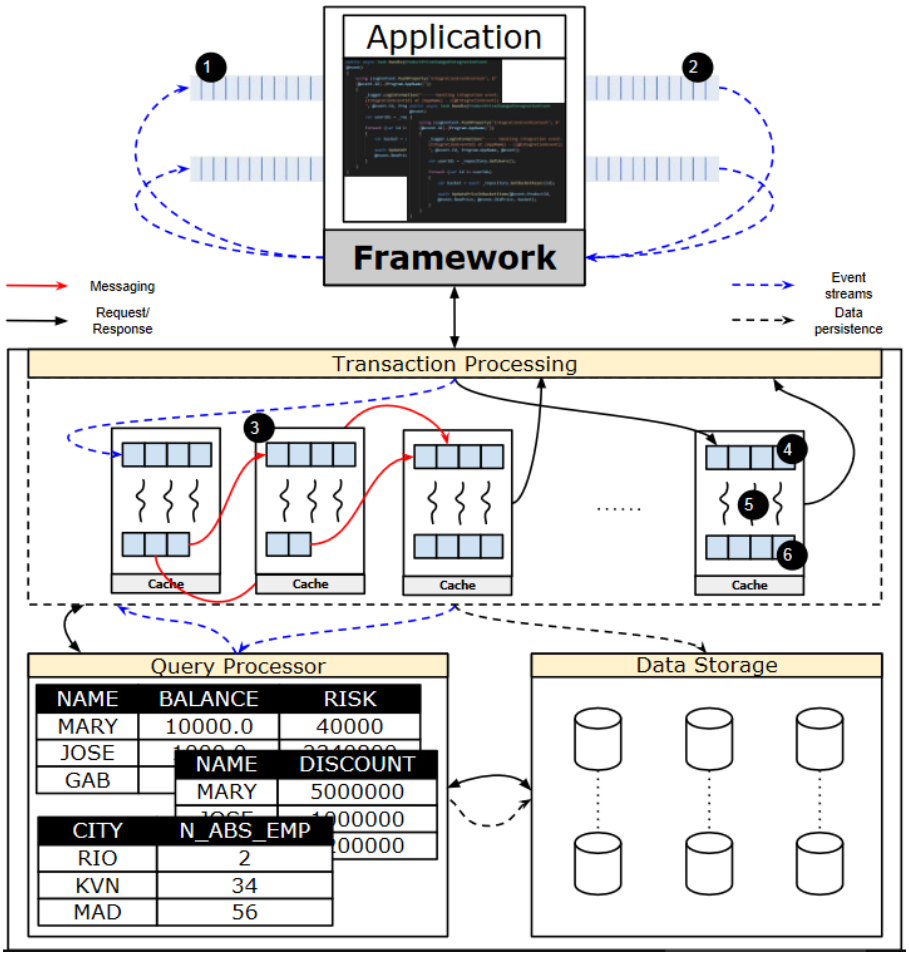


Figure 6.2: A microservice-oriented event-driven database

cation semantics concerning complex data management tasks, but also due to its ability to control the application [88], a necessary condition to enforce event-based constraints and to control the scheduling order of functions.

We envision such a framework should provide programming constructs in sync with the state-of-practice. We use as inspiration the Java Persistence API [186] and declarative transaction implementation in industry-strength web frameworks like Spring [95] to demonstrate our framework constructs, shown in Listing 1, but the approach is generalizable to other programming languages.

In our example, the *Cart* microservice is equipped with a sufficient abstraction to safeguard a given item is correctly locked (lines 2 and 10). The safety guarantee is encapsulated in a *query* directive that is pushed down for database enforcement. In line 18, we exhibit an order dependence, a necessary condition to safeguard consistency guarantees on the total price of an order. The order is enforced both by the database (in terms of concurrent operations) and the framework (at the application level).

```
1  // Stock microservice
2  @Lease("FROM Stock s WHERE s.id = :itemId FOR UPDATE DECREMENT
       s.qtd BY :qtd WITH PERIOD :lease")
3  public Optional<Item> getStockItemInfo(String itemId, int qtd,
       Lease lease);
4
5  // Cart microservice
6  @Inbound(event=AddCartItemRequest)
7  @Outbound(event=AddCartItemAttempt)
8  @Transactional(type="RW", isolation="serializable")
9  public void handleAddCartItem(String customerId, String itemId,
       int qtd) {
10     Optional<Item> item = getStockItemInfo(itemId, qtd,
           Lease.byHours(1));
11     if (item.isPresent()){
12         Cart cart = findCartByCustomer(customerId);
13         cart.add(new CartItem(itemId, qtd));
14     }
15     send(AddCartItemAttempt.build(customerId, qtd,
           item.isPresent()));
16 }
17
18 @Inbound(event=ProductPriceUpdate, precedence=CheckoutRequest)
```

```
19 @Transactional(type="RW", isolation="snapshot")
20 public void handlePriceUpdate(Event productPriceUpdate)
21 // code omitted, handles updates to cart items
22
23 // Order microservice
24 @Query("FROM CustDiscounts cd, Customer c JOIN c.id = cd.c_id
       WHERE c.id = :custId AND cd.disc_id IN (:discounts[id]) AND
       cd.expired() <> FALSE")
25 List findDiscounts(List discounts, String custId);
26
27 @Inbound(event=CheckoutStarted)
28 @Outbound(event=OrderPlaced)
29 @Transactional(type="RW", isolation="serializable")
30 public void processOrder(Checkout checkout) {
31     List discounts = findDiscounts(checkout.discounts,
           checkout.custId);
32     if (discounts.size() != checkout.discounts.size()) {
33         // adjust total price
34     }
35     /* perform necessary data integrity checks, including cart
           items' leases, and build new Order's object */
36     send(OrderPlaced.build(order));
37 }
```

Listing 6.1: Example abstractions provided to the application

Lastly, we demonstrate a compelling abstraction to allow for API-oriented encoding of directives in defined queries. Line 24 shows the API provided by the *User* microservice ("expired()") that encapsulates the business logic regarding the expiration of a discount (defined as a query in the producer side). Flexibility is provided such that data management logic does not need to be hard-coded in the consumer side. The query traverses the Order's microservice without the need to encode error-prone, synchronous calls with weak isolation, since the request is in fact shipped to the database.

Pushing data to the database. We now describe how the encoded data management tasks are recognized and pushed down to the database. In Figure 6.2, a microservice in an application is encoded by its application logic encapsulated in: (a) functions; (b) logical input and output queues (**#1** and **#2**, respectively); (c) invariants (e.g., unique and foreign keys); and (d) external data dependencies.

At start-up time, the framework, by using AOP [2] for example, can proactively identify data management tasks encoded in annotation directives, package them into contextual information representing the microservice, and inform the database. The database then builds an internal representation of the microservice, a virtual microservice, and coordinates with the framework when the register operation has succeed. Then, the framework is able to start to react to and push events, as well as push tasks downward.

6.3.4 Transaction Processing

We now describe the entry point of our database, the transaction processing system, a distributed transaction executor responsible for handling concurrency control and event ordering constraints by managing a virtual representation of microservices. This system targets fulfilling the challenges related to cross-microservice synchronization and validation, event-based constraints enforcement, and strong isolation.

Mapping virtual microservices to computational resources. We envision the database being deployed in both single multi-core machine to be used in small to moderate scale scenarios, and in distributed settings, deployed in a cluster of machines to cope with large scale microservice applications. To support the mapping of computational resources and performance-and-fault isolation, the component will leverage container-based deployments [68]; these need, however, to be adapted to respect virtual microservice isolation boundaries.

As a virtual microservice is a logical isolation abstraction, it requires a mapping to a physical computational resource. As commonly found in microservice deployments [145], containerized microservices provide a compelling abstraction to reason about the mapping of computational resources and performance-and-fault isolation. Thus, container-based isolation makes a promising approach for strong virtual microservice functional isolation sought in microservice architectures, since a virtual microservice naturally inherits the memory space and cores allocated to a container.

Such a scheme allows for reasoning about the distribution of resources in a cluster accounting for resource constraints of each virtual microservice.

While a virtual-microservice deployment scheme will follow the recommendation to separate concerns by mapping one service per container [68], an execution model is necessary. A virtual microservice execution model is composed by the following components:

(a) Input queue of requests (#**4**). Contains both event (sent by other virtual microservices) and transaction requests (i.e., those from clients, do not require a defined event);
(b) Scheduler. A stateful component that processes requests from the input queue. It maintains a mapping between input events and transaction workers.
(c) Transaction workers (#**5**); Perform the heavy-duty work, processing transactions scheduled for execution concurrently with each other. Responsible for atomic execution of operations, including event generation and durability guarantees. As they are stateless, more workers can be instantiated to allow for increased concurrency. The worker messages the scheduler to communicate it has finished its transaction and it is ready for another event processing.
(d) Shared memory. It represents a virtual microservice private mutable state. Besides, in case pessimistic lock-based concurrency control scheme is used, should store the state of the lock table.
(e) Output queue (#**6**). Contains the outbound events, it is, the events scheduled to leave the microservice boundary. Can be formed by both events to other microservices as well as client responses.
Summary. The execution model must fit the key abstraction at the same time allowing for achieving the advanced data management features. Virtual microservices create opportunities to represent a variety of execution models inside the database, but we believe one would not deviate significantly from the exposed here.

6.3.5 Query Processing

We envision a distributed query processor to tackle three principled challenges (§ 6.1): (a) the impossibility of accessing a microservice's private state when the service is down; (ii) to refrain developers from encoding *ad hoc* data replication error-prone mechanisms in the application layer; and (iii) to effectively allow for cross-microservice queries through system-level support, including appropriate consistency guarantees.

Specifically, the strong modularity and the prevalence of short update transactions within a microservice's private state (that may present increased contention) suggests that microservices are better served by specific-purpose transaction workers that share no state and resources across virtual microservices. On the other side, given the ubiquity of accesses to several microservices' private states in online queries, sharing resources while still respecting

the strong isolation principle may decidedly improve performance. The design of the query processing component will balance these competing trade-offs to incorporate the event-based processing and transactional nature of microservices.

Not another query engine. Although multi-tenant databases offer a stronger isolation between tenants, it fails to capture the event-and-data dependencies that are intrinsic across microservices. Therefore, it is necessary to build a different abstraction either on top or apart of query processing engines to support the functional and performance isolation required by microservices.

6.3.6 Data Persistence

This layer is a distributed storage system aimed to store microservices' data and query results. We assume the storage exposes proper interfaces for handling blocks of data that will be processed by the upper layers, namely, the transaction processing and query processing subsystems. The reasoning for a decoupled storage alternative is to allow for flexibility in plugging different storage solutions as well as to offer opportunities to adapt dynamically to varied workloads often found in microservice architectures. We leave further details about data persistence to future work.

6.4 Opportunities and Challenges

In this section, we describe several open questions and promising avenues of future work regarding our architectural vision.

Holistic coordination. Through the virtual microservice abstraction, a dataflow of microservices' interactions can be derived based on the defined inbound and outbound events and the data dependencies that cut across microservices. Treating cross-microservices constraints as event requests within the database opens up opportunities to enforce a processing order across virtual microservices that would conflict otherwise, discharging the virtual microservices to engage in coordination.

Asynchronous code. The abstraction presented in this work expects asynchronous event sending to take place at the end of a procedural function, which is a condition that satisfies most data-intensive microservices [75, 145]. However, we are witnessing an increasing interest in the use of *asynchronous*

function calls that return promises [154]. Asynchronous calls allow for increased opportunities for parallelism and non-blocking application logic inside the database [223]. Investigating the integration of asynchronous primitives for data management logic and virtual microservices is a worthy avenue to pursue.

Application-aware concurrency control. We envision the definition of isolation levels that are most appropriate for the underlying microservice's computation. In our example (Listing 1, line 19), the reasoning is that for client-driven requests, it is assumed that cart items that are already locked face no concurrency issues with requests from other clients, which makes serializable isolation not strictly necessary in this case. At the same time application-defined consistency semantics creates opportunistic window for performance, it creates challenges for concurrency control design within the database.

Query optimization. Microservices are characterized by OLTP-like short update transactions [145] where primary and secondary index lookups solely access its private state are often sufficient, which could be completely handled by the transaction processing component. However, there may be cases where a microservice necessitates querying foreign states, including performing complex aggregation. In this case, such queries could be forwarded to the query processing component to take advantage of more advanced query optimization features. Besides, queries spanning multiple microservices might also benefit from intra-query parallelism enforced consistently by the database. Detecting such cases and coordinating the complex interplay of components is a promising challenge.

Dynamic workloads. Microservices experience a highly dynamic nature. Ideally, microservices should provide the means for growing and shrinking the amount of resources used (e.g., microservices deployed) to cope with varied workloads. Investigating principled approaches, including machine learning techniques [136], to enable dynamic workloads in our architecture is a promising step.

6.5 Conclusion

The data management challenges brought about by microservices necessitate that we rethink the traditional database architecture. With state-of-the-art abstractions, database systems in microservice architectures are unaware

of the significant data flowing outside of the database [112] and hence are condemned to play a secondary role, mostly relegated to only providing data durability.

In this work, we make a case for event-driven microservice-oriented databases. By pushing down virtualized representations of microservices into the database, the database system is able to natively support all benefits pursued by practitioners on adopting microservices, e.g., strong isolation, data ownership, and autonomy, but at the same time offer advanced data management features practitioners currently lack in state-of-the-art database systems.

Realizing this new class of database systems opens up a wealth of research opportunities, including – but not limited to – holistic virtual microservice coordination, asynchronous in-database programming, and application-aware concurrency control.

Chapter 7

vMODB: Unifying Event and Data Management for Distributed Asynchronous Applications

Event-driven architecture (EDA) emerged as an important architectural pattern for designing scalable cloud applications. However, its asynchronous and decoupled nature introduces challenges for meeting ACID properties and data consistency requirements. Database systems, encapsulated and relegated as storage engines of individual components, are unaware of transaction structures cutting across multiple components in EDAs. In contrast, messaging systems are unaware of the components' application states. Weaving such asynchronous and independent EDA components forces developers to give up transactional guarantees, creating data consistency issues. To solve this conundrum, we design vMODB, a distributed framework that facilitates the implementation of highly consistent and scalable cloud applications without giving up the envisioned benefits of EDA. We propose Virtual Micro Service (VMS), a novel programming model that provides familiar constructs to allow developers to specify the data model, constraints, and concurrency semantics of components as well as transactions and data dependencies that cut across components. vMODB leverages VMS semantics to enforce ACID properties by transparently unifying event logs and state management into a common event-driven execution framework. Our experiments using two benchmarks show that vMODB outperforms a widely adopted state-of-the-art compet-

ing framework that only offers eventual consistency by up to 3X. With its high performance, familiar programming constructs, and ACID properties, vMODB will significantly simplify the design and implementation of highly consistent and efficient EDAs.

7.1 Introduction

Event-driven architecture (EDA) emerged as an important architectural pattern for designing scalable cloud applications. An EDA application comprises independent components communicating via asynchronous event messages backed by a messaging system, such as Kafka [137]. An event in EDA serves the purpose of exchanging data, triggering remote computations, and communicating failures. Event producers and consumers in the system are oblivious to each other, promoting loose coupling between them.

The loose coupling of EDA components boosts the independence of component development, allowing different teams to implement and evolve their components with little coordination. As a result, it facilitates the introduction of new components and features, promoting fast-paced changes and supporting business innovation [121]. In addition, EDA brings about many technical benefits, including fine-grained optimization of resource provisioning, elastic scaling of individual components, quicker upgrades and bug fixing, facilitation of feature reuse, more flexibility in integrating with external and legacy systems, etc. [24]. The popularization of EDA has prompted cloud providers to build specific-purpose technologies and adapt their services to support EDA applications. For instance, Amazon AWS offers event messaging services like EventBridge, SNS, and SQS, and workflow orchestration services like Step Functions to facilitate building EDAs [12].

The dominant practice for EDAs employs traditional web application frameworks such as Spring, ASP.NET, and Node.js to implement the application components while relying on messaging systems to enable asynchronous communication [248]. Typically, EDAs follow design patterns resembling the BASE model [204] and adopt eventual consistency. Recent studies [150, 146, 260] and public reports [103, 272, 226, 47, 80, 81, 218, 6, 132, 46, 224, 225, 242, 166, 183, 52] indicate that, with such a practice, developers of EDA systems face many challenges in achieving application safety properties. Being able to execute ACID transactions across multiple components in EDAs would eliminate most of the reported challenges [150].

The technical reasons that EDAs give up ACID include the following. First, event messaging and logging systems such as Kafka [137] are unaware of the progress of transaction execution and the state update operations at the individual components. Therefore, the best achievable outcome is eventual consistency, which relies on message delivery retries and eliminating the impact of duplicate messages [204, 133]. This task per se is complicated in an asynchronous system. Recently emerging distributed application frameworks like Dapr [171] address some of these challenges by offering transparent tracking of event consumption and automatic retries to achieve at least once processing. However, the responsibility for achieving exactly once processing still falls on application developers.

Second, the database systems managing the individual components' states are oblivious to how multiple components exchange events to complete a transaction spanning multiple components [112]. In other words, they are unaware of the global structure of multi-component transactions or the data contained in the events representing the intermediate transaction states. In addition, a database system typically does not allow multiple asynchronous components to interact with the same database within the context of a single ACID transaction. Therefore, database systems cannot achieve ACID properties [28] for multi-component transactions on their own.

Third, existing distributed commit frameworks (e.g., via OpenXA [228]) require writing commit protocol phases explicitly in the application code. Besides being a challenging task for developers, this method deviates from the asynchronous message abstraction and breaks the desired state encapsulation of EDAs. Hence, they are rarely employed in practice in EDAs [150].

In this paper, we argue that, unifying event log and state management within a single system is able to achieve ACID transactional guarantees while withholding the benefits of EDA to application developers. The unification allows the system to be aware of the global structure of transactions, event processing progress, and the state of applications. As a result, the system can reason about the transaction execution status and schedule the processing of events to achieve ACID properties. The unification goes beyond merely adding features of event storage and publish/subscribe APIs to a DBMS, which is inadequate for capturing, from the application semantics, the structure of multi-component transactions necessary to achieve ACID properties and optimize their performance. In other words, the system must provide a programming model to capture sufficient semantic information about the distributed and asynchronous components in EDAs. Realizing this vision

while adhering to EDA principles needs to address several challenges:

- Creating a general programming model to enable functional and state partitioning of applications into independent components.
- Maintaining the principles of state encapsulation of components and asynchronous persistent communication.
- Allowing components to remain autonomous (i.e., as independent deployable units) with independent resource provisioning.
- Capturing the structure of transactions and data dependencies across asynchronous components.
- Ensuring isolation of transactions across multiple components while minimizing coordination.
- Managing data integrity constraints that span across multiple components.

Contributions. To address these challenges, we build vMODB, a novel distributed framework that offers rich features for developing scalable and consistent distributed asynchronous applications.

- We propose Virtual Micro Service (VMS), a programming model that allows developers to specify their application components, including its relational data model, data constraints and dependencies, and input and output event logs (§ 7.2).
- vMODB provides an application framework to realize the VMS programming model. Developers can develop individual application components independently through an ordinary programming language by using the vMODB SDK, which provides the abstraction of data management and event-driven execution (§ 7.2.1).
- Transactions across multiple VMSes can be declared by stating the relevant VMSes and events through transaction graphs. vMODB employs the inversion of control (IoC) principle to control when the VMSes' functions involved in a multi-VMS transaction can be scheduled to ensure transaction isolation (§ 7.2.2).

- VMS semantics are further leveraged to minimize coordination and optimize concurrent transaction execution. vMODB provides programming constructs to allow the specification of the data partition that the transactions will access (§ 7.2.3).
- vMODB contains a purpose-built multi-version database that is aware of the event scheduling order and does not expose in-progress transactions' writes to users. This enables consistent querying of the states of VMSes as well as querying events representing a consistent snapshot of the application state (§ 7.3).
- A scalable coordinator service to schedule transaction execution deterministically is employed by vMODB. The determinism enables various optimizations that maximize performance, including batch commit, asynchronous logging of events, and checkpointing, providing a comprehensive data management solution for applications implemented as EDA (§ 7.4).
- vMODB outperforms state-of-the-art distributed framework solutions that only offer eventual consistency by >3x while coping with ACID properties. Additionally, vMODB offers high scalability in TPC-C, demonstrating its versatility. As a result, vMODB will facilitate the development of highly consistent EDAs without sacrificing performance (§ 7.6).

vMODB is a fully functional system and available as an open source repository: `https://github.com/rnlaigner/vMODB`

7.2 Programming Model

7.2.1 Virtual Micro Services

EDA components are designed in vMODB based on the virtual micro service (VMS) model. The goal of the model is to represent the idea of a fine-grained and independent deployable unit. Although this may remount the idea of the traditional microservice architecture, a VMS is a virtual abstraction managed by vMODB, used to prevent developers from dealing with programming concerns lying outside the application logic, such as network, concurrency, data, storage and log management. These are concerns explicitly handled by practitioners in traditional microservices [150].

A *VMS* contains a state S, a set of application functions F, a map MI, mapping some input event logs to the functions in F, and a map MO, mapping a function in F to an output event log. Each event in the input event logs will trigger each of the functions it is mapped to by MI exactly once. Each function f in F can be mapped by MO to either one event log or none as its event output log. f is an atomic set of actions over S and its output event log specified by MO. The output event log of a VMS function should be unique in the whole system. Besides, there should be no overlapping among event logs in MI and MO in a *VMS*.

VMS uses the relational model for managing states. The state S of a VMS comprises multiple mutable relational tables adhering to a relational database schema. Thus, VMS supports the specification of relational database constraints on the schema, including primary and foreign key constraints, non-null attributes, value-based constraints, cascading deletes, sequences, and more. The set of data items in S can be further divided into 2 non-overlapping subsets: (1) S_N - native tables, managed by this *VMS*. (2) S_F - foreign tables, owned by another *VMS*. S_F can be accessed, but not modified by this *VMS*. More specifically, S_F is a replication of a subset of data items owned by another *VMS*.

VMSes communicate asynchronously via event logs, which follow the EDMA principles to decouple VMSes in time [248]. Each event log is a uniquely identified, append-only sequence of immutable events [113], each being a tuple following a specific event schema. A log supports two operations: (1) *Read*, retrieving an event at a specific offset, and (2) *Append*, appending an event to the log. Unlike VMS states, event logs are not owned by any single VMS. Instead, they are logically shared among VMSes: exactly one VMS has append permission, while the others only have read permission.

VMS System. A VMS system comprises a set of VMSes and a set of event logs. For example, a system containing a *Cart* and a *Product* component can be modeled as two distinct VMSes, as exhibited in Fig. 7.1. The *product* VMS manages the product information, and the *cart* VMS manages the state of the user carts. Additionally, *cart* replicates some product information managed by the *product* VMS, a recurring pattern in EDA to reduce latency [146]. Through the event log identified as `update-price`, prescribed by its mapping MO, *product* VMS appends updates to its products' prices. This makes the update available for *cart* VMS consumption, and upon reception, the function prescribed by its mapping MI is triggered, updating the product replica. In practice, these operations enabled through event logs tend to cut

across many components in EDA systems [161]. Therefore, it is desirable that multi-component workflows enjoy transactional semantics.

Transaction Graphs. In order to compute the dependencies of multi-VMS transactions, a transaction graph has to be declared for each. In the graph, each vertex is a VMS function, and each edge is an event log. In other words, a transaction in vMODB consists of a set of functions of a group of VMSes and a set of read/append operations carried out on a set of logs defined in the corresponding transaction graph.

Trending cloud application architectures and programming models that empower EDAs, such as microservices and FaaS, often present multi-component interactions that form topologies [123, 280]. Topologies are usually known in advance or simply captured by tracing mechanisms [161]. To minimize coordination, vMODB incentivizes the developers to formulate their transactions as topologies. Having a cycle in the graph requires extra coordination to ensure isolation. For instance, in a cyclic topology, component `A` could consume two or more different event logs at different steps of the same transaction. In such a scenario, all subsequent transactions would remain blocked until all the operations of the transaction on `A` are completed, negatively impacting the system's performance.

Key Insight. The VMS model specifies the structures of an EDMA application, components' behaviors, the state and event dependencies, and precise system guarantees. This not only provides developers with tools to define and reason about the architecture and behaviors of their applications, but also enables the underlying system to identify potential bugs during compilation, such as incorrectly specified VMS mappings or updates to replicas of data owned by other components. In contrast, existing popular platforms, such as Dapr and other composite platforms, impose little semantics on the application architecture and hence are unable to support these.

7.2.2 Programming with Virtual Micro Services

This section presents the programming constructs offered by vMODB SDK to developers. The running examples use Java but the constructs utilized are transferable to other programming languages.

7.2.2.1 Building Blocks

Listing 1 exhibits an excerpt of the Cart component. As shown in lines 1-14, tables are specified by annotating plain objects with the `@VmsTable` annotation. Table columns are derived from attributes annotated with `@Column` or `@Id`. Both declare an attribute as a column in the table, while the latter indicates that the column is a part of the primary key. Primary key indexes are created automatically, and secondary indexes can be declared on arbitrary columns, as seen in lines 3 and 5, a composite secondary index on `seller_id` and `product_id`. Users can also declare foreign keys on other VMS tables, as observed in line 7, and are automatically indexed by vMODB. Optionally, users can declare constraints on columns, such as seen in line 11, on which a voucher assigned to an item cannot be negative.

An instance of a class annotated with `@VmsTable`, such as `Cart Item`, abstracts one row through an update operation, as seen in line 28. Common database operations are abstracted by the `IRepository` interface (Line 15), which includes traditional operations such as `lookupByKey`, `insert`, `delete`, and `update`. Users can optionally extend this interface to provide additional pre-declared queries, such as the one seen in line 16. Queries can also be declared programmatically in the method body via a query-building API. It is worth noting vMODB transparently selects the available index via an optimizer. For example, in this case, the product_idx index (declared in lines 3 and 5) is selected by vMODB to execute `getProductByKey` query (line 25).

A VMS is declared by annotating a class with `@Microservice` (line 21), and event-driven functions of a VMS are declared by the annotation `@Inbound`. The `@Inbound` annotation and the parameter of the method reflect which event log triggers the method and the corresponding event type. Users must indicate a name to facilitate the event log identification, as observed in method `updateProductPrice` (line 23). There should be one `IRepository` interface per VMS table. Instances of `IRepository` interfaces are created dynamically and made available to functions transparently via dependency injection technique [87]. Thus, users can write any arbitrary application logic within a VMS function as well as query and update the application state by using `IRepository` facilities. Declaring output events is optional, and its absence indicates no events are generated by the VMS function.

```
@VmsTable(name="cart_items")
public class CartItem {
    @Id @VmsIndex(name = "product_idx")
    public int seller_id;
    @Id @VmsIndex(name = "product_idx")
    public int product_id;
    @Id @VmsForeignKey(table = "user", column = "customer_id")
    public int customer_id;
    @Column
    public String product_name;
    @Column @PositiveOrZero
    public float voucher; // other attributes below...
}
interface IProductReplicaDB extends IRepository<ProductReplica> { }
interface ICartItemDB extends IRepository<CartItem> {
    @Query("select * from cart_items where seller_id = :sellerId
        and product_id = :productId")
    List<CartItem> getCartItems(int sellerId, int productId);
}
@Event
record PriceUpdated(int sellerId, int productId, float price) { }
@Microservice("cart")
public class CartVMS {
    @Inbound("price-updated") @Transactional(type = RW)
    public void updateProductPrice(PriceUpdated priceUpdated) {
        var product = productReplicaDB.getProductByKey(
            priceUpdated.sellerId, priceUpdated.productId);
        product.price = priceUpdated.price;
        productReplicaDB.update(product);
        List<CartItem> items = cartItemDB.getCartItems(
            priceUpdated.sellerId, priceUpdated.productId);
        for (CartItem item : items) {
            item.voucher += (priceUpdated.price - item.unit_price);
            item.unit_price = priceUpdated.price;
        }
        cartItemDB.updateAll(items);
    }
}
```

Listing 7.1: Excerpt of Cart component as a VMS

A VMS function can be annotated with `@Transactional` (line 23) to enable transactional support. Optionally, users can specify whether such a function operates on the VMS state in read-write (RW) or read-only (R) mode, allowing for vMODB optimizations. Unspecified application functions are treated as RW. As seen in `updateProductPrice` method, no begin-transaction or commit statements are necessary. In the case of a constraint violation, the transaction is aborted without user intervention. vMODB transparently manages transactional concerns, allowing for clean application logic specification.

7.2.2.2 Composing Multi-VMS Transactions

```
1  @Microservice("product")
2  public class ProductVMS {
3      @Inbound("update-price") @Outbound("price-updated")
4      @Transactional(type = RW)@PartitionBy(method = "getId")
5      public PriceUpdated updateProductPrice(PriceUpdate
           priceUpdate){
6          Product product = productDB.lookupByKey(
7              priceUpdate.seller_id, priceUpdate.product_id);
8          product.price = priceUpdate.price;
9          product.updated_at = priceUpdate.dt;
10         productDB.update(product);
11         return new PriceUpdated(priceUpdate.sellerId,
12             priceUpdate.productId, priceUpdate.price,
13             priceUpdate.version, priceUpdate.instanceId);
14     }
15 }
16 @Event
17 record PriceUpdate(int sellerId,int productId,float price,Date dt)
18 { (int,int) getId(){ return new (sellerId,productId); }  }
```

Listing 7.2: Excerpt of Product component as a VMS

The output event log of a VMS function can be the input event log of another VMS, composing the functionalities supported by multiple VMSes (§7.2.1). We illustrate this via the *Product* component exhibited in Listing 2. To start, not all events in a VMS system are necessarily produced by VMSes. Some user-facing events can be specified to trigger a VMS function based on a user request. For example, the input event `update-price` in line 3

corresponds to a user request to update the price of a given product, thus triggering the method `updateProductPrice` in line 5. User-facing events enjoy the same log mechanisms as internally generated events. The *updateProductPrice* function generates the event `PriceUpdated` (line 13). As observed in the method declaration, the output event `price-updated` matches the input event of Cart in Listing 1, line 23. vMODB automatically identifies event dependencies across VMSes and transparently schedules them without any user-specific implementation other than the VMSes' application logic.

The composition of multiple VMSes via event dependencies serves as the building blocks for declaring a transaction spanning multiple VMSes. Listing 3 shows how to use vMODB's API to construct a transaction topology across *Cart* and *Product* VMSes composed of the functions triggered by `update-price` and `price-updated` events, respectively. Every transaction is identified by a name (line 1) and starts with an input event (line 2). In this case, as only two VMSes participate in the transaction, the specification soon finishes with a terminal node (line 3). A terminal node indicates that no further events will be included in the transaction guarantees, even though the terminal component generates output events whose processing would not have transactional guarantees.

```
1  var updatePriceTx = TransactionBootstrap.name("update-price-tx")
2            .input(alias: "a", vms: "product", event: "update-price")
3            .terminal(vms: "cart", dep: "a");
4  var checkoutTx = TransactionBootstrap.name("checkout-tx")
5                  .input("a", "cart", "customer-checkout")
6                  .internal("b", "stock", "reserve-stock", "a")
7                  .internal("c", "order", "stock-confirmed", "b")
8                  .internal("d", "payment", "invoice-issued", "c")
9                  .terminal("e", "seller", "invoice-issued", "c")
10                 .terminal("f", "shipment", "payment-confirmed", "d")
```

Listing 7.3: Registering multi-VMS transactions

Real-world EDAs often require many components to interact (§ 7.2.1). Line 5 presents the `checkout-tx` specification, which reflects a transaction topology of six components. In this transaction, four components (`cart`, `stock`, `order`, and `payment`) generate events for downstream processing. Differently from `updateprice-tx`, `checkout-tx` contains two terminal nodes, indicating that both components must conclude the event processing successfully in order to commit the transaction. Besides, each terminal can have

different event dependencies, as seen in lines 9 and 10. In sum, through the vMODB API, users specify which events (i.e., edges) require ACID transactional guarantees.

7.2.3 Parallelization Modes

To reduce coordination further, vMODB enables a transactional function of a VMS to operate in two additional modes: partitioned and embarrassingly parallel modes. While the default single-thread mode is designed for transactions that require RW access to the entire state, the partitioned and parallel modes allow for concurrent access to the VMS state. In partitioned mode, RW functions can exclusively access specific partitions based on keys, while in parallel mode, R functions or RW functions performing only insert operations can access any data concurrently. Developers can choose the execution mode so that isolation is not jeopardized. Since functions that need RW access to the entire state, where key-based access cannot be defined, are often exceptions rather than the norm, we anticipate that developers will utilize the different execution modes to optimize the concurrency of VMSes.

To allow developers to specify the concurrency modes of VMS functions, vMODB prescribes two annotations, `@PartitionBy` and `@Parallel`, for key-based and embarrassingly parallel executions, respectively. For instance, in Listing 2, line 5, the method utilized to determine the key-based access is "getId," found in the input event class `PriceUpdate` (line 18). This means all data accesses this function performs will relate to the particular product ID. For parallel access, simply annotation a function with `@Parallel` is sufficient and indicates the function can run concurrently on multiple input events. An incorrect specification can create conflicts, which will be detected, and the conflicting transactions will be aborted to achieve safety. Details are discussed in § 7.4.1.

7.2.4 Discussions

7.2.4.1 Black-box Applications

A primary issue with current EDA practice is that the application is a black box and does not expose semantic information to the database system to enforce data consistency and integrity. vMODB addresses this challenge by opening up the black box and unveiling the semantic information of trans-

actions and data operations via the VMS model. The VMS model allows developers to define data constraints and dependencies of transactions at the application level. vMODB can enforce transactional properties and data constraints transparently and globally, thanks to its cross-stack design that spans the application, messaging, and database layers.

Compared to existing programming models of distributed frameworks such as Dapr [171], vMODB does not require more exposure to internal data models of individual components. In addition, although vMODB requires developers to specify the cross-component transaction topologies, the specification requires similar information as the workflow management features in frameworks like Dapr. Thus, vMODB does not create a higher level of coupling or dependencies than existing frameworks. On the other hand, vMODB achieves ACID properties, while existing frameworks only guarantee at least once processing at best.

7.2.4.2 Familiar Programming Constructs

Despite the ambitious goal of vMODB, we endeavor to lower its adoption barrier and employ programming abstractions widely adopted by developers. The input/output mapping of events to application functions through metaprogramming is prevalent in the SDKs of existing distributed frameworks. VMSes are modeled based on the service [92] and repository [91] patterns, popular in database-backed applications [249]. The data access API extends the Java Persistence API [186] (JPA) with familiar concurrency constructs and prevents users from managing connections, sessions, transactions, and locks as traditionally in JPA implementations. Overall, VMS promotes a loosely coupled design as found in EDAs, so there is no direct remote communication among VMSes.

7.2.4.3 Comparison to Actors

Actors share some similarities to VMSes. Both adopt the EDA principles, though, in a different way. In actor frameworks, each actor can be considered a fine-grain component in an EDA application, and it only has a single-threaded execution mode. Developers using actor frameworks must partition their applications into fine-grain actors, which do not share states. Determining the granularity of actors and how to partition the applications into actors imposes complexities to system design for the developers, which often

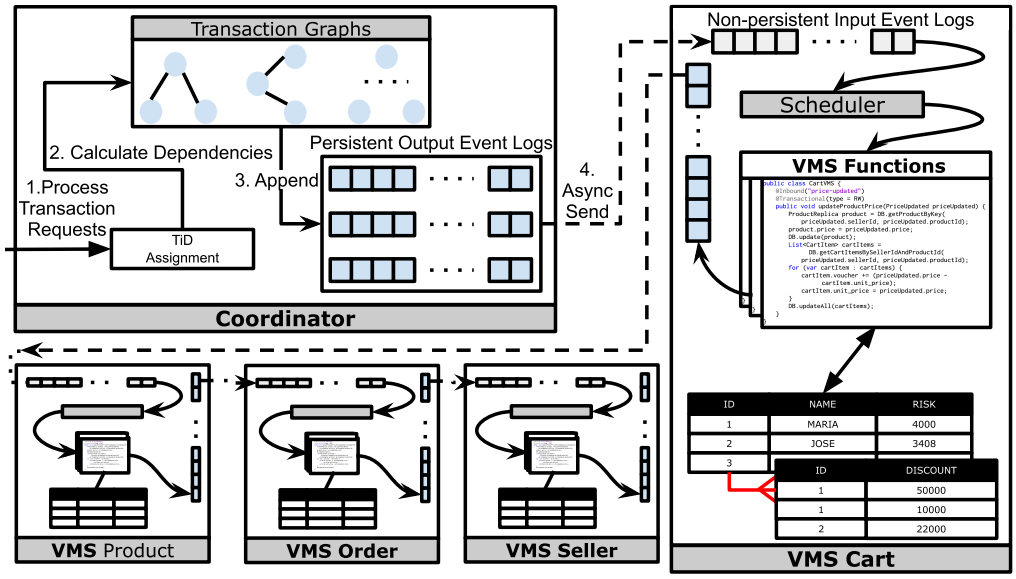


Figure 7.1: Architectural Overview of vMODB

involves trade-offs between performance and data consistency [265]. Furthermore, if the workload involves transactions across multiple actors, there is a high penalty to the performance of these transactions [73, 157]. This is because any multi-actor transaction would become distributed transactions even if the involved actors are, for example, co-located in the same container. On the contrary, VMS is at the granularity of deployment units, i.e., one VMS corresponding to one container. It encapsulates a larger state than an actor with a rich relational data model. It exposes multiple event-driven functions that can have different concurrent execution modes. Furthermore, transactions involving only data within a deployment unit (i.e., a VMS) can be executed efficiently as non-distributed transactions.

7.3 Architectural Overview

In this section, we present an overview of vMODB, a distributed framework that unifies event logs and state management for distributed asynchronous applications through the VMS programming model. vMODB aims to provide users with full control over the allocation and management of computational resources. This design goal matches the current state of practice, where

virtual machines or containers are used to execute cloud services, allowing software teams to reap benefits associated with on-demand resource provisioning, fine-grained scalability, and high modularity of EDAs. vMODB is entirely implemented in Java 21 with 35K lines of code.

VMS Instances. As shown in Figure 7.1, the vMODB architecture is organized as a collection of VMS instances, which can be independently deployed in different machines or share the resources of particular machines, allowing for great flexibility without changing the application code. Each VMS instance is an operating system (OS) process that employs different threads for varied tasks. Independently of the deployment model, a VMS instance, by being an OS process, operates in its own isolated memory space. Therefore, VMSes are isolated from each other and do not share states.

Coordinator. The coordinator plays a central role in the system, processing transaction requests from clients, computing the dependencies of each VMS in a multi-VMS transaction, assigning monotonically increasing TiDs to transaction requests, dispatching transaction requests with an assigned TiD to input VMSes, and coordinating the commit protocol. The coordinator is also an OS process; thus, it can be deployed in an existing or a standalone node in the system.

State Management. The state management module of a VMS instance manages the relational tables, executes data operations, and enforces data constraints. It maintains multiple versions of the state updated by transactions and uses a transaction's TiD to determine which state version a transaction should operate. It relies on the ordered TiDs scheduled to assign versions to data items and a deterministic transaction execution scheme to ensure isolation. This multi-version design allows readers to access a consistent snapshot concurrently with other transactions without blocking.

Event Log Management. The dominant practice of event log management is relying on external systems such as Kafka, which creates performance and consistency challenges in EDAs [150]. vMODB removes the indirection between event log systems and application components by unifying event log management, data management, and event-driven execution into a common framework. This design decision is key to reap application design and performance advantages. The producer is responsible for persisting their logs and making them accessible. While the events are pushed from the producer to the consumers for normal transaction execution, vMODB also provides a pull API for transaction abort and recovery.

Transaction Scheduler. The transaction scheduler of a VMS instance

uses a deterministic transaction execution scheme [157] to determine which VMS functions should be executed based on the available events in the input event logs, the TiD of the events, and the concurrency hints of the VMS functions. This design characterizes the Inversion of Control (IoC) [87] principle, in which the custom-written application codes do not control the execution; rather, the framework runtime controls how and when VMS functions must execute. Note that this design is important to safeguard the serializability of multi-VMS transactions and does not contradict the sovereignty of components. Software teams are still in control of the computational resources of the VMSes, and only the internal execution schedule is controlled by vMODB. In other words, user threads that do not interfere with transaction scheduling can be executed independently, such as read-only queries.

7.4 Protocols

Achieving ACID transactions across components in an EDMA is challenging due to asynchrony and data decentralization. Existing implementations of distributed commit protocols, such as 2-Phase Commit, are not only costly but also exhibit blocking behavior, requiring waits among participating components and typically using communication outside event channels, thereby jeopardizing asynchrony and decoupling.

To solve this conundrum, we propose an efficient deterministic scheme that preserves the asynchronous message communication abstraction and does not violate the sought-after state encapsulation principle of EDMAs. It is based on the key observation that the topology of multi-component transactions of an EDMA application is deterministic and known in advance [123, 280]. However, it does not assume that the VMS functions are deterministic, contrasting the assumptions in deterministic databases [251].

7.4.1 Transaction Management

vMODB divides transaction scheduling into two parts to maximize performance. The coordinator reasons about transaction dependencies at the VMS level to determine the execution order across VMSes. Next, individual VMSes exploit semantic information from user specifications to maximize concurrent access to data items without violating the transaction order defined by the

coordinator. Different from state-of-the-art deterministic protocols [251], the read-write set of transactions is not required at any point in vMODB.

7.4.1.1 Coordinator Transaction Scheduling

The coordinator assigns a unique and ordered TiD to each multi-VMS transaction in batches. In Figure 7.3, each transaction request is processed by a TiD Assigner worker, responsible for assigning it a TiD and computing its precedent TiD on each VMS it involves, according to the transaction specification. The latter is needed because not every transaction involves all VMS instances. To enable scalable transaction submission while preserving the monotonically increasing property of TiD assignment, we divide the TiD namespace into non-intersecting, fixed-size ranges. Workers are placed in a ring, each operating over an exclusive range. TiDs are assigned in the context of a batch. A batch is sealed when a worker runs out of TiDs or a parameterized epoch elapses. Then, the worker with the smallest *id* sends the last assigned TiDs of each VMS to the next worker in the ring, which then merges what it receives with its local information and passes it to the next worker in the ring. Upon receiving the information from its previous worker, it has sufficient information to reason about the precedent TiD of each transaction on each VMS. So it can emit the transactions and start processing the next batch. This information-forwarding process for each batch stops when every worker has heard from the previous worker in the ring. Furthermore, the workers also assign the current batch ID to each emitted transaction. Workers are given a batch ID to start, and when a batch is sealed, they compute the next batch ID as the current batch ID plus the number of TiD assigner workers.

Algorithm 1 exhibits the ring formation procedure. It starts by defining the starting TiD, which can be 1 in a brand-new system or the last committed TiD (line 1). In the former case, the VMSes TiD precedences are all set to zero. In any case, the VMSes TiD precedences serve as the local information dependence for the first worker in the ring (lines 2-7). Next, it stitches the local info dependency queues across subsequent workers (lines 9-21), spawning each worker to operate in a unique subset of TiDs (line 16). In other words, the output queue of a worker's local information becomes the input queue of the subsequent worker. Lastly, it continuously probes for transaction requests and dispatches them to a randomly chosen worker, effectively spreading the workload seamlessly across them (lines 22-24).

Algorithm 1: Coordinator Transaction Scheduling: Workers Setup

Input: $numWorkers$ (number of TiD assigner workers)
Input: $nextBatchId$ (next batch identifier, 1 by default)
Input: $lastTidCommitted$ (0 by default)
Input: $lastTidPerVms$ (a dictionary, $\emptyset$ by default)
Input: Δ (epoch duration)
Input: $maxTxnsPerBatch$ (max number of transactions per batch)
Input: $VMSes$ (set of VMSes)
Input: $txnDefs$ (a dictionary with transaction definitions)
Input: Infinite stream of transaction requests $\mathcal{R} = (r_1, r_2, \ldots)$

$nextTid \leftarrow lastTidCommitted + 1$;
if $lastTidPerVms = \emptyset$ **then**
    for $vms \in VMSes$ **do**
        $lastTidPerVms.add(key : vms, value : (lastTid : 0, lastBatch : 0, prevLastBatch : -1))$
$firstWkrLocalInfoInput \leftarrow \emptyset$;
$firstWkrLocalInfoInput.queue(lastTidPerVms)$;
$currWkrLocalInfoInput \leftarrow firstWkrLocalInfoInput$;
$idx \leftarrow 1$;
do
    $currWkrLocalInfoOutput \leftarrow \emptyset$;
    if $idx = numWorkers$ **then**
        $currWkrLocalInfoOutput \leftarrow firstWkrLocalInfoInput$;
    $wkr \leftarrow buildWorker(idx, numWorkers, nextBatchId, nextTid,$
        $\Delta, maxTxnsPerBatch, txnDefs,$
        $currWkrLocalInfoInput, currWkrLocalInfoOutput)$;
    $spawn(wkr)$;
    $nextBatchId \leftarrow nextBatchId + 1$;
    $nextTid \leftarrow nextTid + maxTxnsPerBatch$;
    $currWkrLocalInfoInput \leftarrow currWkrLocalInfoOutput$;
    $idx \leftarrow$ idx + 1;
while $idx \leq numWorkers$;
for $txReq \in \mathcal{R}$ **do**
    $wkr \leftarrow$ pick a random worker;
    $wkr.queue(txReq)$;

Algorithm 2: Coordinator Transaction Scheduling: Worker Event Loop Part 1

Input: id (worker identifier)
Input: $numWorkers$ (number of TiD assigner workers)
Input: $initBatchId$ (initial batch ID in this worker)
Input: $initTid$ (initial TID in this worker)
Input: Δ (epoch duration)
Input: $maxTxnsPerBatch$ (max number of transactions per batch)
Input: $VMSes$ (set of VMSes)
Input: $txnDefs$ (a dictionary with transaction definitions)
Input: $localInfoInput$ (previous worker local info queue)
Input: $localInfoOutput$ (next worker local info queue)
Input: Infinite stream of transaction requests $\mathcal{R} = (r_1, r_2, \ldots)$
Output: Groups of ordered input events $\mathcal{I} = (e_1, e_2, \ldots)$

```
1  batchCxt = (batchId : initBatchId, initTid : initTid);
2  nextTid ← initTid;
3  while true do
4      lastTidBatch ← nextTid + maxTxnsPerBatch − 1;
5      endTs = now() + Δ;
6      while nextTid ≤ lastTidBatch ∧ now() < endTs do
7          if ((r = R.poll()) <> null) then
8              ProcessTxnReq(r);
9      if NoProgress() then continue ;
10     for each transaction input that is pending the local info from the
         previous worker in the ring, update its lastTid and dispatch it;
11     update precedent batch ID and last TiD of each VMS with the
         local info from the previous worker in the ring;
12     seal and multicast batchCxt to terminal VMSes;
13     if numWorkers > 1 then
14         nextTid ← initTid + (numWorkers ∗ maxTxnsPerBatch);
15     initTid = nextTid;
```

Algorithms 2 and 3 exhibit the TiD Assigner worker functionality. Algorithm 2 starts by defining the current batch context based on information received from the setup procedure (line 1), and then attempts to process as many transactions as possible within the maximum batch epoch and the maximum number of transactions per batch (lines 6-8).

In Algorithm 3, the `ProcessTxnReq` function begins by obtaining details of the transaction graph, specifically the source VMS and all participant VMSes (lines 2-3). Next, it loops over the participant VMSes to verify whether each VMS requires receiving local information from the previous worker in the ring, and adds it to the pending set if so (lines 8-11). Still within the loop, it updates the lastTid and the number of TiDs in this batch for this VMS. Next, it stores the terminal VMSes in the transaction (in order to identify which ones must participate in the commit protocol, explained later in Section 7.4.1.3). Finally, if the current transaction event has pending VMSes, it is cached to be submitted later, once the local information from the previous worker in the ring arrives. Otherwise, it dispatches the transaction event to the source VMS.

Back to algorithm 2, after exhausting the number of TiDs or batch epoch, it checks whether there has been actual progress in the ring by calling the function `NoProgress` in algorithm 3. The function verifies whether any transaction has been processed. If so, it returns false, indicating that progress has been made. Otherwise, it checks whether it is a single worker and if no local information from a previous worker is available, returning true in either case. Lastly, it checks whether the local information received from the previous worker in the ring matches the local information sent by this VMS, indicating that no progress has been made since the last batch processed by this VMS.

Returning to algorithm 2, line 10 incorporates the local information from the previous worker in the ring into the pending transaction events (line 11, algorithm 3), and then dispatches them. Line 11 updates the batch ID, last batch ID, and last TiD for all VMSes, and determines the number of TiDs per VMS in the batch. Line 12 seals the current batch with the above information, forming a $batch\_commit\_info$ message, and propagates it to the terminal VMSes in this batch. Finally, lines 13-15 update the next TiD according to the number of workers in the ring.

Algorithm 3: Coordinator Transaction Scheduling: Worker Event Loop Part 2

```
1  ProcessTxnReq(r):
2      sourceVms ← txnDefs[r.txnName].source();
3      VMSesInTxn ← txnDefs[r.txnName].VMSes;
4      prevTidPerVms ← ∅;
5      pendingVMSes ← ∅;
6      for vms ∈ VMSesInTxn do
7          prevTidPerVms.add(vms.identifier, vms.lastTid);
8          if vms.batchId <> batchCxt.batchId then
9              vms.batch = batchCxt.batchId;
10             vms.numberOfTIDsBatch = 0;
11             pendingVMSes.add(vms.identifier);
12         vms.lastTid = nextTid;
13         vms.numberOfTIDsBatch+ = 1;
14     batchCxt.terminals.add(txnDefs[r.txnName].terminals());
15     if pendingVMSes <> ∅ then
           // cache transaction input pending local info
16     else
           // dispatch r to sourceVms with
              batchCxt.batchId, nextTid, prevTidPerVms
17         ;
18     nextTid+ = 1;
19 NoProgress():
20     if nextTid = initTid then
21         if numWorkers = 1 then return true ;
22         if localInfoInput = ∅ then return true ;
23         else
24             prevWkrLocalInfo ← localInfoInput.peek();
25             return whether the prevWkrLocalInfo is the last local
                  info sent over localInfoOutput in this VMS;
26     return false;
```

7.4.1.2 VMS Transaction Scheduling

Within each VMS instance, multiple versions of each data tuple are maintained. We use S to represent the set of state versions, with S_0 representing the initial VMS state. By default, a VMS function triggered by an event runs in a single-threaded mode. It will finish processing an event before processing the next. For events involved in multi-VMS transactions, their processing order is scheduled according to the TiDs. To eliminate unnecessary coordination, we provide several programming constructs for developers to give hints to vMODB to minimize coordination. A function of a VMS can be declared as read-write (RW) or read-only (R). After an RW function processes an event belonging to a transaction with $TiD = x$, the state is updated to S_x *iff* it is a valid state, i.e., no constraint is violated during function execution. RW functions never need to wait for events triggering an R function with a smaller TiD to arrive. This is because an event that triggers an R function with $TiD = y$ can read the state version S_z, where S_z is the latest version with $z < y$.

User-provided concurrency hints can further optimize vMODB transactions by allowing independent transactions to modify the database state concurrently without coordination. Through a key-based access specification, developers specify a key-based partition of the state on which a given function operates exclusively. Through an embarrassingly parallel access specification, developers specify that a given function will only perform insert operations. Read-only queries operate in an embarrassingly parallel manner by default. These conditions guarantee that concurrent state updates converge naturally to a valid state.

A transaction manager module ensures that transaction isolation is maintained during partitioned or parallel execution. If a conflict is detected, the TiD is aborted. This mechanism protects against an incorrectly specified execution mode by developers.

Algorithms 4 and 5 present the algorithms used by the VMS transaction scheduler to schedule transaction events for execution. It divides transaction scheduling in a VMS into three functions, *Ingest*, *Schedule*, and *OnTxnCompletion*, which are executed by different user-level threads concurrently.

In Algorithm 4, the *Ingest* procedure prepares transaction inputs for scheduling by reading all available input events from the inbound event stream and building transaction contexts for execution. A transaction context maps an input event to the respective application function of a VMS.

In Algorithm 5, the *Schedule* procedure continuously tracks the execution of transactions and makes a best effort to schedule as many concurrent transactions as possible while respecting the global order defined by the coordinator. More specifically, in line 5, it retrieves the next transaction to execute based on the ID of the last executed transaction (*lastTidExec*) and blocks if no new transaction can be scheduled.

If the transaction event triggers a single-threaded function, it first checks whether partitioned or parallel transactions are running and blocks until that condition is no longer true. Next, it checks whether the transaction runs as an R function in this VMS (i.e., a read-only execution). If so, it dispatches it for concurrent execution by a thread pool. This is possible because the multi-version database allows a subsequent function to run concurrently with this R function, as explained above. Otherwise, it executes the transaction inlined. Since this transaction must be the next *lastTidExec* to advance transaction scheduling, executing inlined avoids thread context-switch costs.

Transaction events triggering embarrassingly parallel functions (lines 12–14) are dispatched concurrently to a thread pool. In addition, events triggering partitioned functions (line 15) accessing different partitions, or accessing the same partition as the previously dispatched events, but all of the latter are R functions (line 16), can be dispatched to a thread pool (line 18). Furthermore, it disallows transaction events triggering functions with different modes to execute concurrently unless all the existing transactions are read-only (lines 7, 12, and 15). The thread blocks if no events can be scheduled (lines 8 and 20) to release resources for the other threads to carry out other tasks, e.g., processing events, query execution, and checkpointing (§ 7.5).

Finally, the *OnTxnCompletion* procedure in Algorithm 4 is a callback executed by the thread assigned to execute a transaction function. It updates *lastTidExec* and the sets *partSched* (the set of partitioned transactions scheduled) and *paraSched* (the set of parallel transactions scheduled), allowing the scheduler (*Schedule*) to proceed with further transaction scheduling. In the event of an error, it triggers the abort procedure, as described in Algorithm 6.

Correctness Argument. The correctness of the VMS scheduling algorithm is analyzed as follows. It is based on the assumption that users provide a correct specification of application function execution modes.

Theorem 7.1. *Let $O = [T_1, T_2, \ldots, T_n]$ be a total order of transactions defined by the coordinator, and S be the schedule produced by the VMS. S is conflict-equivalent to O.*

Algorithm 4: VMS transaction scheduling - Part 1

Input: Infinite input event stream $\mathcal{I} = (e_1, e_2, \ldots)$
Input: $lastTidCommitted$ (0 by default)
Output: Infinite output event stream $\mathcal{O} = (o_1, o_2, \ldots)$

```
1  txns ← ∅;// transaction context set
2  lastTidExec ← lastTidCommitted;
3  Ingest():
4      while true do
5          I' ← pollAllBlocking(I) where I' ⊂ I;
6          for e ∈ I' do
7              f ← mapVmsFunction(e);
8              txnCtx ← buildTxnCtx(e, f);
9              txns ← txns ∪ txCtx
10 OnTxnCompletion(txn, status):
11     if status = ERROR then
12         abort(txn);
13         return;
14     lastTidExec ← max(lastTidExec, txn.Tid);
15     txn.isFinished ← true;
16     O ← O ∪ txn.result;
17     if txn.mode = PARALLEL then
18         paraSched ← paraSched \ txn;
19     else if txn.mode = PARTITIONED then
20         partSched ← partSched \ txn;
```

Algorithm 5: VMS transaction scheduling - Part 2

```
1  paraSched ← ∅;// set of parallel transactions scheduled
2  partSched ← ∅;// set of partitioned transactions scheduled
3  Schedule():
4      while true do
5          txn ← txn′ ∈ txns : txn′.prevTid = lastTidExec;
6          if txn.mode = SINGLE_THREAD then
7              if ¬(partSched ≠ ∅ ∨ paraSched ≠ ∅) ∨ (∃t : t ∈
                partSched ∪ paraSched ∧ t.type = RW) then
8                  block while above cond. holds;
9              if t.type=R then dispatch(t); else execute(t);
10             continue;
11         do
12             if txn.mode = PARALLEL ∧ (partSched = ∅ ∨ (∀t : t ∈
                partSched ∧ t.type = R)) then
13                 paraSched ← paraSched ∪ txn;
14                 dispatch(txn);
15             else if txn.mode = PARTITIONED ∧ (paraSched =
                ∅ ∨ (∀t : t ∈ paraSched ∧ t.type = R)) then
16                 if partSched[t.part] = ∅ ∨ (∀t : t ∈
                    partSched[t.part] ∧ t.type = R) then
17                     partSched ← partSched ∪ txn;
18                     dispatch(txn);
19                 else
20                     block until partSched[t.part] = ∅;
21             else break ;
22             nextTxn ← txn′ ∈ txns : txn′.prevTid = txn.Tid;
23             execModeAux ← txn.mode;
24             txn ← nextTxn;
25         while txn ≠ ∅ ∧ execModeAux = txn.mode;
```

Proof. The VMS scheduling algorithm ensures that transactions with different execution modes {Single threaded, Partitioned, Parallel} are not scheduled concurrently. It places transactions into groups, with all transactions in a group belonging to exactly one of the three execution modes. Transactions in each group have TiDs that are greater than all the TiDs of the transactions in any preceding groups. Furthermore, transactions in each group are fully executed before any transactions in subsequent groups, except for read-only (R) transactions. The algorithm ensures that all R transactions are dispatched after all potentially conflicting RW transactions with smaller TiDs have been executed. Therefore, the VMS multi-version database ensures the latter reads the version according to the schedule stated in O. In the following analysis, we only focus on RW transactions.

For each transaction group, there are three possible cases:

Case (i). All transactions are single-threaded transactions. The VMS scheduling algorithm ensures that for any pair of single-threaded RW transactions, they are executed strictly according to the order in O without concurrency.

Case (ii). All transactions are partitioned transactions. Let $G(O)$ be the conflict graph of the transactions in this group induced by the total order O, and $G(S)$ be the corresponding conflict graph induced by the schedule S.

Let $T_i \rightarrow T_j \in G(O)$. This implies:

- T_i and T_j conflict.
- $T_i \prec T_j$ in O.

The algorithm ensures that RW transactions accessing conflicting partitions are never scheduled concurrently. Therefore, for any pair of conflicting transactions T_i and T_j such that $T_i \prec T_j$ in O, the conflicting operations from T_i precede those from T_j in S. Hence, the edge $T_i \rightarrow T_j$ exists in the conflict graph $G(S)$, preserving the conflict-equivalence with $G(O)$.

Case (iii). All the transactions are embarrassingly parallel transactions. As we assume that they have no conflicting access to the VMS state, $G(O)$ of the transactions in this group does not have any edges. Therefore, the corresponding $G(S)$ does not violate any precedence relationship in $G(O)$.

All in all, the complete schedule S is conflict equivalent to the schedule stated by O. □

7.4.1.3 Batch Commit

Committing every transaction in a distributed setting introduces substantial network round-trip times between participant nodes and a transaction coordinator [176]. To mitigate this overhead, a transaction runs and commits in the context of a batch of transactions in vMODB. When a batch is sealed, the coordinator collects all VMSes that are terminal nodes in the batch and sends a *batch_commit_info* to each, including the batch ID, the previous batch, and the number of TiDs for each. This procedure is described in Section 7.4.1.1 and shown in Algorithm 2.

Every VMS instance maintains a count of the number of TiDs executed per batch. Upon receiving a *batch_commit_info* message, a terminal VMS instance checks whether it has already processed all TiDs of the given batch. Otherwise, it waits until all TiDs complete processing. Then, it sends a *batch_complete* message to the coordinator. When all *batch_complete* messages arrive, the coordinator then sends a *batch_commit* request to all other participant VMSes (i.e., non-terminals) in the batch.

Acknowledging that a batch has completed in terminal VMSes implies that upstream VMSes have also completed. This insight allows for minimizing messages in the batch commit protocol. Besides, transactions are optimistically processed, that is, VMSes do not wait for batch commit completion in order to process transactions of subsequent batches.

7.4.1.4 Transaction Abort

The abort procedure exhibited in Algorithm 6 is explained as follows. If a VMS function fails to comply with a database constraint or an incorrectly specified execution mode causes a conflict, the affected VMS starts the abort process. It (i) stops scheduling new transactions, (ii) removes the aborted transaction from its set of transactions, (iii) cleans up the write sets of the transactions with TiDs between the aborted TiD and the last executed TiD, and informs the coordinator about the abort. The coordinator logs the abort information (TiD, batch ID) and multicasts it to VMSes involved in the corresponding transaction (only upstream VMSes are necessary). Upon receiving an abort message, VMSes proceed similarly to the VMS that started the abort procedure, executing steps (i) to (iii) above. VMSes must ignore incoming input events with TiD > aborted TiD until the producer VMS restores the precedence of input events. Transaction scheduling is resumed

from the successor of the aborted transaction in the log. Note that at this point, VMS states have not been checkpointed yet, and the event logs written as part of the current batch execution have not been logged yet. The protocol is largely decentralized. It is only necessary to involve the coordinator because upstream VMSes may be unknown to the VMS that initiates the abort procedure.

Algorithm 6: VMS transaction abort

```
   Input: txn that led to a conflict or constraint violation
   Input: Infinite input event stream 𝓘 = (e_1, e_2, ...)
   Input: Transaction context set txns
   Input: Set of VMS tables T
 1 block (I);// block Ingest polling I
 2 block (txns);// block Schedule access to txns
 3 if (txn abort is raised by this VMS) then
 4     forward abort msg to Coordinator;
 5 txnsToAbort ← ∀txn' ∈ txns : txn'.Tid ≥ txn.Tid;
 6 txns ← txns \ txnsToAbort;
 7 for table ∈ T do
 8     for txnToAbort ∈ txnsToAbort do
 9         table.versions ← table.versions \ txnToAbort.writeSet;
10 if (VMS is the first node in the transaction graph) then
11     log txn abort event in VMS state;
12     find the successor and predecessor events of txn aborted in I;
13     adjust dependencies of transaction events;
14     reinsert events after txn aborted in I;
15 else
16     find the predecessor event of txn aborted in I;
17     wait until the successor event of the predecessor is sent through
         I;
18 lastTidExec ← predecessor.Tid;
19 unblock (I);
20 unblock (txns);
```

7.4.2 Durability

Logging. While SQL statements act as the logical logging constructs for traditional DBMSs, event logs are leveraged as the logical logging in vMODB. Event logs trigger computations in application components and, in consequence, updates to their internal states. Thus, it is advantageous to leverage the event log rather than the possibly many individual updates to VMS states, allowing for minimizing I/O overhead. The logs dispatched to consumer VMSes are placed into a pending logging buffer. Unlike using an external traditional event log system designs [137], vMODB only needs to persist logs at commit time. The pending logging buffers are flushed upon a batch commit to amortize disk I/O overhead.

Checkpointing. Logical logging alone can lead to a prohibitive cost on state recovery because all event logs require replaying to align the VMS states to reflect the last committed batch. Therefore, vMODB features a checkpointing mechanism that is performed at batch commit time. A background thread in each VMS identifies which data item versions are part of the batch pending commit and flushes them all to disk atomically. The process can optionally perform garbage collection, removing old versions that no new transaction will "see" to reduce memory pressure. By counting on the multi-version VMS database, the checkpoint process does not conflict with any concurrent function or require synchronization with other modules.

7.4.3 Fault-Tolerance

Assume that vMODB components can crash at any time, resulting in the coordinator and VMSes losing their data in memory.

VMSes. The VMSes can replicate the event logs to standby replicas for failure recovery. In this case, sender I/O threads, besides logging log segments, can stream them to replicas. The replicas process the log segments in the same way as the primary but do not send output events to consumer VMSes. For failure detection, replicas rely on heartbeats received from primary nodes. Upon a detected failure, replicas trigger a leader election. Similarly to Raft [184], replicas start as candidates and rely on the number of votes received to decide whether the candidacy is successful. In this case, it broadcasts the election result. As transactions execute optimistically (§ 7.4.1), a newly elected leader may not contain some log segments. Therefore, it is necessary to request possible missing log segments from upstream VMSes.

Upon receipt, the system can return to normal operation.
Coordinator. To make the coordinator fault-tolerant, upon all *batch_complete* messages' arrival, the coordinator also sends a *batch_commit* message to standby replicas. In this case, committed batch IDs serve as the unit of replication. Upon election, the new coordinator requests the last committed batch ID from all VMSes (VMSes metadata are also streamed from coordinator to replicas) to cover cases where the coordinator crashes before replicating it. Upon receiving the responses, it updates its last committed batch ID if necessary and sends a *batch_aborted* message to all VMSes. VMSes then proceed similarly to a transaction abort. VMSes discard non-committed data item versions, and TiD precedences are adjusted accordingly to receive transactions from the new coordinator node. Differently from VMSes, the coordinator does not stream event logs to replicas since these are simply transaction requests that can be retried by clients. Although this may lead to losing some useful work the VMSes performed, it avoids a TCP round trip to each replica for every transaction emitted.

7.5 Implementation and Optimizations

7.5.1 VMS Internals

To provide a high-performance framework that unifies event and data management, numerous optimizations are required, including network I/O, transaction scheduling, concurrent data access, query processing, and storage formats for both data and event logs. Figure 7.2 illustrates the VMS internals.

7.5.1.1 I/O Processing

Network communication with the coordinator and among VMSes is achieved with TCP connections through asynchronous I/O [185]. This applies to both event logs and protocol messages. I/O threads are divided into two categories, receivers and senders. The OS kernel signals receivers about new network I/O events, which in turn unmarshal log entries from the buffers, deserialize individual input events, and queue them for transaction processing. Whenever output events are made available, senders dispatch them to the corresponding consumer VMSes. If there is a failure on sending event logs (e.g. VMS crash or network partition), they are put in a pending buffer

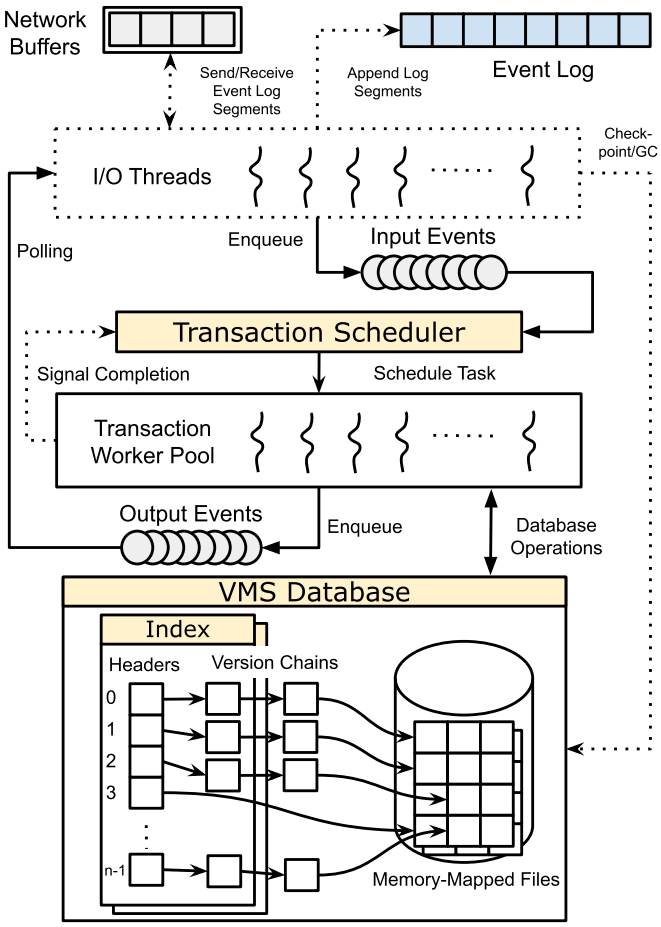


Figure 7.2: VMS Internals

for later reattempt.

A configurable number of receivers are placed in a network thread pool, returning whenever network events are not available. Senders, in turn, run independently and are uniquely assigned a VMS consumer to send log entries to. This design prevents cache pollution, that is, network events cache-lines evicting VMS consumer cache-lines, and vice versa. In this case, whenever output event logs are not available, senders yield their CPU time slice to other system threads.

7.5.1.2 Transaction Management

A transaction scheduler thread is responsible for scheduling VMS functions for execution. It reasons about the optimization hints provided by the developers in the VMS application code (§ 7.2.3) to maximize throughput while coping with transaction isolation. The transaction scheduler maintains a work-stealing pool of transaction worker threads responsible for executing VMS functions. The work-stealing pool enhances performance because once multiple functions are scheduled to execute, there are no dependencies between them, and, therefore, transaction worker threads are free to "steal" tasks from each other, preventing additional thread parking and unparking costs. To maximize resource efficiency, the transaction scheduler thread blocks if no new event input is available for scheduling. Upon terminating a function execution, the transaction worker thread updates the scheduler TiD progress, queues the event output, if present, to sender I/O threads, and proceeds to steal a task from the pool.

7.5.1.3 State Management

vMODB provides a purpose-built database to minimize coordination and maximize VMS concurrency. The specialization is based on the following:
Natively in-process. A in-process VMS database is instantiated for each VMS instance in vMODB to manage its relational tables. Albeit state-of-the-art databases like Duck-DB [72] operate in process, applications typically necessitate costly calls to database running outside the application process for data access (e.g., via JNI [188]). In vMODB, as all components are natively in process, that is, calls never leave the application process context, allowing data transfers between the VMS database and application logic with minimal overhead. In addition, data access technologies like ORMs through JDBC [187] must parse application objects into a database message protocol and vice versa. vMODB, however, operates natively with entities, speeding up key tasks such as constraint enforcement, query processing, and durable writes.
Multi-versioning. To optimize the execution of concurrent transactions and the processing of ad hoc read-only queries, VMS database is designed as a multi-version system. Differently from traditional multi-version systems that assign transaction identifiers or transaction timestamps in-flight [269], VMS database relies on the ordered TiDs assigned by the coordinator to

assign versions to data items and therefore preserve the illusion that each transaction is executing alone on a dedicated system.

The use of a specialized multi-version system for a VMS is beneficial for several reasons: (i) Read-only transactions that do not generate output event logs do not need to be scheduled using the same (thread/CPU) resources allocated to read-write transactions; besides, since its execution does not jeopardize the system progress, its execution can be scheduled at any time as long as the respective versions are available to be read, increasing scheduling flexibilities of the system. (ii) Even if a read-only transaction generates an event output log, the multi-versioning allows read-only transactions to always be scheduled as embarrassingly parallel, maximizing performance. This would not be possible in a lock-based mechanism. (iii) Apart from multi-VMS transactions, users may issue ad hoc queries to a VMS instance. The multi-version database enables users to query a consistent snapshot of the VMS state at any moment.

Lock-free data structures. VMS database employs lock-free in-memory data structures for primary and secondary indexes, allowing transaction worker threads to read and create new versions of data items simultaneously without coordination. Tables are currently implemented as a collection of concurrent hash tables – single-value hash tables for unique indexes (e.g., primary key indexes) and zero or more multi-value hash tables for secondary indexes. vMODB supports range queries iterating over primary or secondary index entries. Extending the interfaces to tree structures [106] is left for future work.

The different versions of data items are kept in primary key indexes, and secondary indexes only maintain a pointer to the associated primary index entry. Data item versions are chained as a linked list in a primary key index entry. The head represents the latest data item version. Each node in the list contains the associated TiD representing its version. The versions in the linked list are naturally ordered by vMODB's concurrency control mechanisms.

Query processing. To take further advantage of the native in-process design and decrease pressure on memory caches, vMODB packages with several predefined query execution plans that employ operator inlining to optimize query execution. If a query does not have a predefined query plan, vMODB can switch to the traditional volcano strategy [106]. In addition, different from ORM libraries that are oblivious to query plans, vMODB integrates both query execution and entity construction into a common execution plan.

As a multi-version system, entities are naturally cached and reused to reduce the costs of serializing object results.

Data persistence and version deprecation. A VMS database operates in two modes: in-memory and checkpointing. The former only stores data in memory, while the latter uses a memory-mapped file (MMF) maps to the last data item versions. vMODB leverages the in-process, multi-versioning, and lock-free design for additional optimizations. As explained later (§ 7.4), flushing (fresh) and removing (deprecated) data item versions can proceed safely with concurrent transactions and ad hoc queries.

7.5.2 Coordinator

The coordinator bootstraps with a handshake protocol. Users must specify a set of transactions (Listing 3) and corresponding "starter" VMSes, including their network addresses. At startup, the coordinator sends a presentation message to all VMSes and requests their VMS metadata, which includes their input and output event logs. With this information, the coordinator builds a consumer set for each VMS instance, that is, the VMS instances they must connect to. After these steps, the coordinator and VMSes are ready to process transactions. Additionally, the coordinator provides an HTTP server so HTTP clients can submit transaction requests.

I/O processing in the coordinator follows a similar design as a VMS, but is targeted at maximizing transaction request dispatching while not jeopardizing the TiD assignment process. To meet this goal, sender I/O threads never block. Instead, in the absence of new event inputs, they yield their CPU time to another thread. This ensures minimized waiting times for VMSes because blocking would lead the I/O thread to park and unpark, a costly process for input generation. Receiver I/O threads function in the same way as in a VMS. TiD assigners assign TiDs to multi-VMS transactions and compute the dependencies of each VMS involved. They operate as independent threads in an event loop mode (i.e. non-blocking), outside the context of a thread pool.

The main thread processes batch complete messages from terminal VMSes and sends out batch commit requests to remaining participating VMSes. It also ensures the required VMSes receive the abort information in case of an abort.

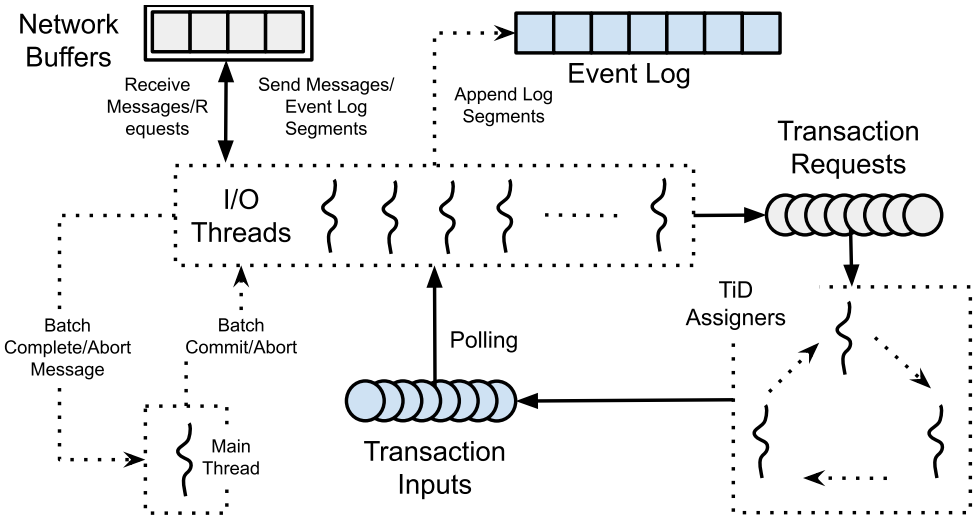


Figure 7.3: Coordinator Internals

7.5.3 Event Log Management

vMODB applies several optimizations to reduce I/O overhead that applies to both the coordinator and VMSes. Unlike traditional event log systems, on which clients are offered iterator-based (often blocking), interfaces for reading and appending to the log [137], in vMODB, the consumption and production of event logs are handled transparently from the user code. As a result, custom-written application logic does not need to block waiting for and publishing events.

To decrease the amount of bytes sent over the network, I/O threads serialize the payload of events generated by VMS functions into log entries using a configured serialization format (e.g. compacted JSON). To decrease the number of I/Os, the log entries are batched into segments to fit an operating system (OS) page before being sent to the associated VMS consumers. For cache-friendliness, the network buffers are also sized to an OS page.

Each segment is mapped to a region of memory allocated by the OS (i.e. native memory regions), allowing efficient data transfers between the application and the kernel. These regions of memory are cached and reused across I/O threads. Entries are appended contiguously within the segment and never overflow the page, avoiding page segmentation.

Sender I/O threads employ best-effort batching, i.e., event logs are batched

within a segment as much as possible, but the priority is submitting events as fast as possible, so Nagle's algorithm [175] is disabled. The goal is to achieve high throughput without compromising latency, a good trade-off found in the experiments (§ 7.6). Whenever network channels are busy/saturated, the sender I/O threads switch to logging mode and switch back when the network channel is free. This design allows for higher resource usage and efficiency since it prevents the creation of additional threads for logging tasks.
Log Segment Design. A segment of the log is a subset of event logs that follows a compact structure, composed of: *i*) segment size in bytes (int); *ii*) number of event logs (int)[1-N]; and *iii*) a sequence of event log entries. An event log has the following format: *i*) message type (byte); *ii*) batch (long), TiD (long), and event identifier (string); *iii*) event payload size (int); *iv*) event payload (byte)[1-N]; *iii*) dependence map size (int); and the *vi*) dependence map payload, the precedence of TiDs for VMSes involved in the transaction.

As events flow downstream, the VMSes no longer involved in the transaction are removed from the dependence map, allowing for decreased metadata. The offset of an event in the log is the TiD assigned by the coordinator. Concurrent transactions in VMSes, although never leading to corrupted states, can lead to out-of-order appends to the log. However, this is beneficial because sender I/O threads do not need to coordinate with transaction workers, allowing for higher performance. This is possible because every VMS guarantees that the events logs are processed in TiD order.

7.5.4 Discussions

Contrasting to the conventional tiered architecture of data-intensive applications [18], vMODB manages application and state holistically to reap the several optimizations discussed above while maintaining the appealing simplicity of a stateless program design for developers. vMODB employs Inversion of Control (IoC), which makes event processing and function scheduling transparent to users as in other distributed frameworks, while enforcing transaction isolation. The design of vMODB also takes advantage of the substantial increase in main memory sizes observed in recent years. As these large memory sizes are often sufficient to accommodate the entire state of transactional systems, EDAs with partitioned components [204] can also benefit from it.

We took extra care while designing the VMS internal components towards extensibility. VMS transaction and log management and storage APIs

are modularized to allow for future endeavors in disaggregated storage [197]. While it is possible to follow this line and design the VMS database APIs as a thin layer on top of other database systems, as long as they expose transactional and relational APIs and version metadata (for concurrent writers support), this would come at the cost of jeopardizing some advancements discussed in § 7.5.1.3.

7.6 Evaluation

7.6.1 Experimental Settings

7.6.1.1 Benchmarks

We use two benchmarks throughout the experiments: Online Marketplace (OM) [146] and TPC-C [55]. OM models an e-commerce application consisting of seven EDA components: Cart, Product, Stock, Order, Payment, Shipment, and Seller. OM prescribes key data management requirements and properties in real-world EDA deployments, including distributed transaction processing, data replication, data and event querying and processing, and data integrity constraints. Our experiments use the workload from OM [146], with sellers querying their orders and adjusting their products to maximize profits online and customers managing their carts and requesting checkouts. We model a high-peak-order processing scenario, which allows stressing the performance of the systems due to the higher complexity of the `Customer checkout` transaction. `Price update`, `Product delete`, and `Query dashboard` correspond to 10% each and `Customer checkout` corresponds to 70% of the transaction ratio.

To demonstrate the versatility of vMODB, we also use the standard OLTP benchmark TPC-C [55]. To port TPC-C to vMODB, we partition TPC-C tables into three components: (i) *Warehouse* embodies the customer, district, and warehouse tables; (ii) *Inventory* holds the item and stock tables; and (iii) *Order* is responsible for order, order item, and new order tables. Similar to previous work [157, 251], our evaluation uses the `New Order` transaction. Thus, we decompose the transaction logic of `New Order` across the components in association with their tables without violating application semantics, a design that resembles a real-world partitioned distributed application [204, 150]. Tables in TPC-C are typically partitioned based on the warehouse.

We use this as a hint for specifying VMS concurrency primitives, which is discussed next.

7.6.1.2 Implementations

For experiments with OM, we select Dapr [171] as a baseline system, a choice stemming from several reasons: (a) Dapr is the most popular open-source framework for building distributed and event-driven applications [59] and widely used in industry settings [171]; (b) Dapr meets key EDA principles as vMODB, prioritizing independent deployment units and loose coupling among components; (c) Dapr imposes no particular programming model that necessitates fine-grained partitioning of states to application developers; (d) Dapr provides transparent event log management to application developers; (e) Dapr follows an eventual consistency model, usually linked with higher performance in partitioned designs [204, 133], a model widely adopted in industry [150].

For a thorough comparison, we design two different Dapr versions:

(i) Following a traditional microservice style deployment [150], each component is designed as a stateless service, offloading all data management operations to its particular DBMS. We chose PostgreSQL due to its widespread popularity and support for concurrent writers.

In addition to the default PostgreSQL setting ("Dapr+PG Default"), we developed an optimized "Dapr+PG Tuned" implementation that includes ORM optimizations and tuned PostgreSQL parameters to maximize performance. In the application ORM, we set the max connections to 10K, bypassing the default limit of 100, and we configure a connection pool to reuse connections. Besides, we disable synchronous commit, so updates are eventually logged rather than atomically[1] We use Unix domain sockets for inter-process, kernel-based communication instead of the default TCP sockets. We increase the pgsql buffer size to improve analytical query performance. We increase the shared buffer from 128MB to 3GB, which corresponds to 15% of 21 GB available in c5n.2xlarge instance, as suggested by PostgreSQL documentation. The work mem is set to 128 MB instead of the default 4MB.

(ii) For a direct comparison with vMODB, we design a relational in-memory database library for Dapr that mimics PostgreSQL APIs. The reason

[1]In off mode, there is no waiting, so there can be a delay between when success is reported to the client and when the transaction is later guaranteed to be safe against a server crash. (The maximum delay is three times *wal_writer_delay*.)

Table 7.1: VMS concurrency primitives used in Online Marketplace

Component	**Transaction**	**Primitive**
Cart	Customer Checkout	Keyed on `customer_id`
Product, Cart, Stock	Product Delete	Keyed on `seller_id, product_id`
Product	Price Update	Keyed on `seller_id, product_id`
Cart	Customer Checkout	Keyed on `customer_id`
Order	Customer Checkout	Keyed on `customer_id`
Payment	Customer Checkout	Parallel
Shipment	Customer Checkout	Parallel
Seller	Seller Dashboard	Parallel

Table 7.2: VMS concurrency primitives used in TPC-C

Component	**Primitive**
Warehouse	Keyed on `warehouse_id, district_id`
Inventory	Keyed on `warehouse_id`
Order	Parallel

is that Dapr's default state management library relies on a key/value API. In a traditional Dapr deployment, the key-value database is deployed on a separate machine, often shared across components, which can jeopardize performance and access isolation. Our library has the performance advantage of cache locality and offers a relational data model. It supports linearizable updates for single tuples with lock-free operations for higher concurrency, but does not provide transactional isolation. This is in line with the default consistency model offered by Dapr. The in-memory database library uses write-ahead logs ("WAL" in the plots), whose updates are flushed to disk in batches periodically by a background job.

Dapr ships by default with Redis Streams [211] as its event log system [67]. We use default Redis to obtain a baseline through a pure in-memory execution and Redis with durability enabled to log event logs ("EL" in the plots).
VMS Parallelization. Table 7.1 exhibits the VMS concurrency primitives used in Online Marketplace and Table 7.2 exhibits for TPC-C. In Dapr, all operations run as a parallel task given the lack of transaction isolation.

7.6.1.3 Deployment

We conduct all the experiments on Amazon EC2 instances deployed in the eu-north-1 region. Unless explicitly noted, each component mentioned by the experiment is assigned a particular EC2 instance to avoid resource competition. We use C5n instance types because they are ideal for network-intensive applications [10]. Each instance runs Ubuntu 22.04.4 with Linux kernel 5.10.17 and hyper-threading enabled.

The OM components are written in C# for Dapr and Java for vMODB. We use Dapr .NET SDK and runtime version 1.11.1, and .NET 7.0.119. The .NET SDK is a flagship Dapr SDK, being the first SDK made available and the most popular [60, 61]. Each component is deployed in c5n instances (large, xlarge, 2xlarge) across different experiments. Redis and vMODB coordinator are each deployed in an x9large instance. TPC-C experiments do not include Dapr because it lacks ACID properties and relational constraints.

7.6.1.4 Methodology

All experiments are run over 6 epochs of 10 seconds each, with the first 2 epochs used to warm up the systems. After each epoch, the components' states and event logs are cleaned. To remove cache effects across experiments, we restart OM's components after each experiment.

We collect two metrics in the experiments: throughput and end-to-end latency. End-to-end latency is measured as the interval from when a transaction is emitted by the workload driver to when the driver receives the corresponding batch completion information from vMODB. In Dapr, a special event (emitted by the last component in a transaction) is used to acknowledge the end of a transaction. Given the event-driven nature of the systems, the results are received asynchronously.

We extend the OM benchmark driver to support experiments in both Dapr and vMODB. Unless explicitly mentioned, the vMODB coordinator is configured with 8 network threads, a TiD Assigner thread, and a background thread that asynchronously notifies committed batches to the driver. By default, the epoch size is set to 500 ms with a maximum of 10K transactions per batch. When either option applies, the coordinator seals the batch.

By default, in OM, VMSes are configured with half the number of vCPUs as transaction workers and the other half as I/O threads. This is due to better resource balancing for both transactions and ad hoc queries.

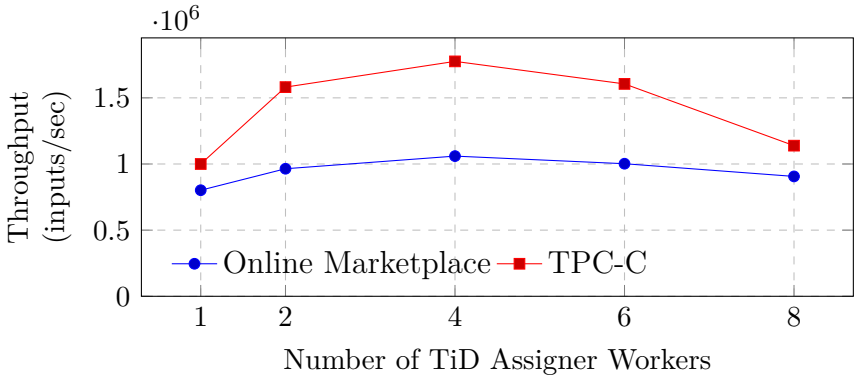


Figure 7.4: Throughput of TiD Assignment Varying Number of Workers

For TPC-C, we match the number of transaction workers in each VMS and the number of client threads submitting transactions to the number of warehouses to accompany the scale factor. Additionally, transaction workers execute in the common worker pool, with the JVM responsible for assigning transaction tasks to common pool worker threads.

7.6.2 Transaction Scheduling

We design a microbenchmark to characterize the performance of vMODB TiD assignment across different benchmarks. This set of experiments was run on a high-end laptop with an Apple M1 chip, 8 cores (4 for performance and 4 for efficiency), and 16 GB of RAM. We pre-load the system with enough transactions to prevent TiD assigners from running out of work. For OM, we reuse the transaction ratio specified in § 7.6.1.1. Since the seller dashboard is a read-only query that does not require a TiD, we allocate its percentage uniformly across other transactions.

In the first experiment, we fix the batch epoch to infinity and the maximum number of transactions per batch to 100k. Figure 7.4 shows that increasing the number of TiD assigner workers can provide increased throughput. TPC-C shows higher scalability due to the reduced number of VMSes, resulting in lower overhead on computing dependencies across workers. In any case, though, the wait time incurred by waiting for dependencies on the ring increases above four workers.

We also verify whether the maximum number of transactions per batch affects TiD generation performance, since a larger batch size can mask the

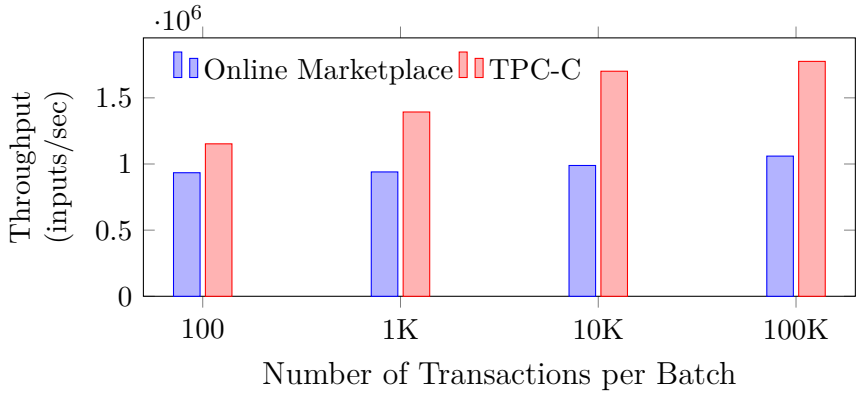


Figure 7.5: Throughput of TiD Assignment Varying Batch Size

wait time introduced by the "ring". In this experiment, we fix the number of TiD assigner workers to 4 and vary the batch size. We observe in Figure 7.5 that the batch size indeed has an impact. For OM, a batch size of 100K improves throughput by 8,8% compared to 100. Greater improvements are observed in TPC-C.

The experiments highlight the efficiency of the coordinator design. A single worker generates workloads (with nearly 1 million requests/second) that can stress state-of-the-art OLTP DBMSs and are sufficient for a vMODB system. As a result, the coordinator cannot be the bottleneck for a useful vMODB setup.

7.6.3 Online Marketplace

7.6.3.1 Effect of Concurrency Level

In this experiment, we refer to *concurrency level* as the maximum number of concurrent transactions running in the system at a given time. Apart from capturing the sensitivity to varied concurrency degrees, this experiment is useful to identify the parameter that yields the optimal performance in each system for subsequent experiments. To this end, we maximize the resources available to each OM component, and we set the workload skewness to be uniform for both Seller and Product.

As shown in Figure 7.6, vMODB shows increasing throughput as the concurrency level increases. The maximum throughput is achieved at the highest concurrency level (36), but from 28 on, the throughput stabilizes as

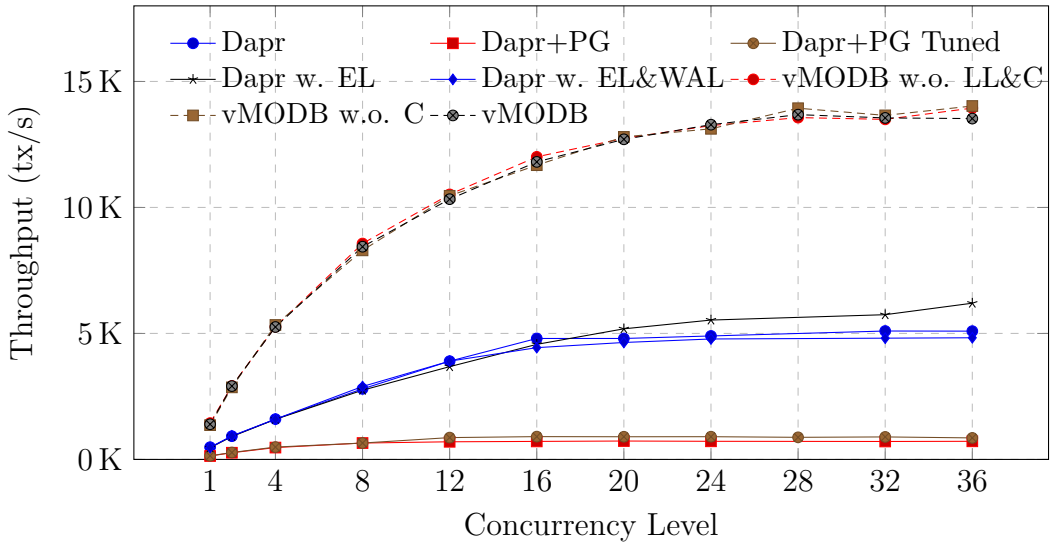


Figure 7.6: Concurrency Overhead

the system is saturated. On the other hand, we observe a substantial performance difference in Dapr. Starting with the PostgreSQL-based solutions (Dapr+PG), we found that even tuning Npgsql (the ORM) and PostgreSQL did not improve performance. The workload is dominated by I/O between the OM components and PostgreSQL, including inter-process communication, object serialization/deserialization, and connection management, and it exhibits the lowest throughput across the experiments. Since PostgreSQL and Npgsql are not natively part of Dapr, we do not include them in subsequent experiments.

With the in-memory execution, the I/O overhead found in Dapr+ PG is dramatically reduced. The remaining sources of overhead in Dapr can be explained by two factors: the sidecar design [287] and the remote event log system. The sidecar is an autonomous process that mediates external messages to and from a Dapr component, resulting in creating several threads in addition to the application threads, leading to an intensive inter-process communication and context switch overheads, not to mention the intensive network I/O to read and append event logs.

Breakdown Latency. We also collect the latency breakdown of the OM transactions. Figure 7.8 exhibits the numbers collected with the highest con-

currency level. Being a long-running and complex transaction, the `Customer Checkout` latency in Dapr stands out substantially. A similar trend is found in other frameworks for cloud-native applications [146, 156]. `Product Delete` and `Price Update` transactions involve fewer components and are less prone to increased latency. vMODB latencies remain stable due to the batch commit protocol, which we explore further in § 7.6.3.4. Lastly, `Seller Dashboard` exhibits lower latency in vMODB, demonstrating the beneficial aspects of the VMS database (§ 7.5.1.3) over the native C# query operators [173]. It is worth noting that Dapr provides no transaction isolation, whereas vMODB offers users a consistent snapshot for queries.

Effect of Durability. The logical logging and checkpointing mechanisms impose little overhead on both platforms. In vMODB, logical logging ("LL" in the plots) adds 3,5% overhead and checkpointing ("C" in the plots) adds another 3,2%, demonstrating the effectiveness of optimizing for I/O in vMODB design (§ 7.5.3). On the other hand, from concurrency level 24 on, Dapr shows better performance with logical logging enabled. The reason is the use of multi-thread I/O to accelerate reading and writing to I/O sockets, so more cycles are devoted to accessing data [210]. This is only observed when persistence is enabled. An in-depth investigation of this trend is out of the scope of this paper.

7.6.3.2 Scalability

In this experiment, we aim to compare the scalability of Dapr and vMODB. We focus on the `Customer Checkout` because it is the most complex transaction in OM, imposing the highest coordination overhead [156, 146]. To this end, we vary the amount of resources available to each OM component and maximize the concurrency level. Similar to warehouses in TPC-C, the partitioning unit in OM are sellers. Thus, customers pick items from a specific seller in this experiment. Figure 7.7 shows both systems scale as more resources are added, with vMODB being superior in every instance, even ensuring ACID properties. Moreover, vMODB, with limited resources, is competitive with Dapr when full resources are available, demonstrating the effectiveness of the overhead-minimization strategies adopted (§ 7.2.3).

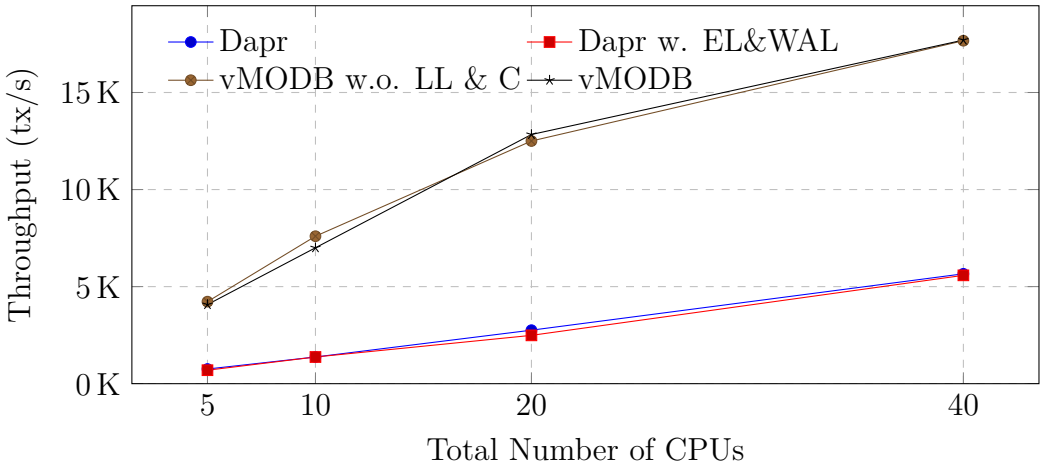


Figure 7.7: Scalability

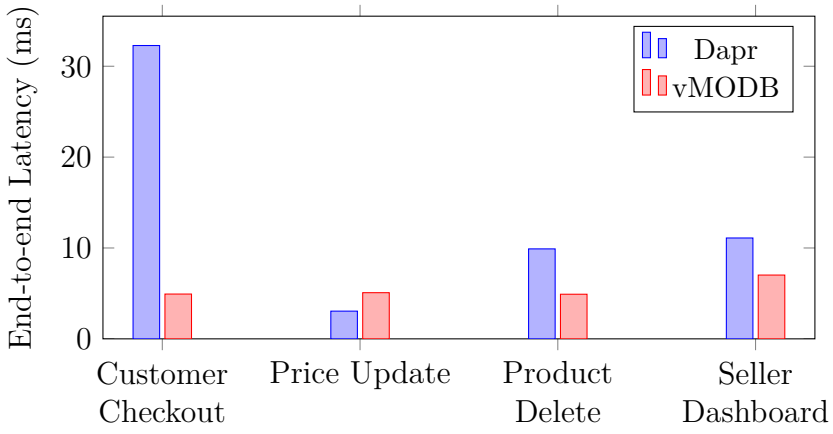
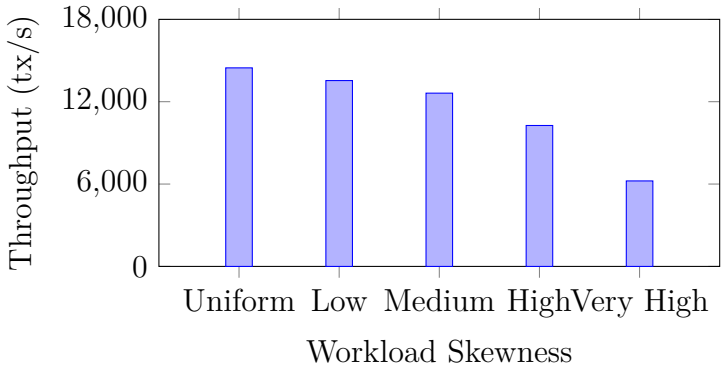
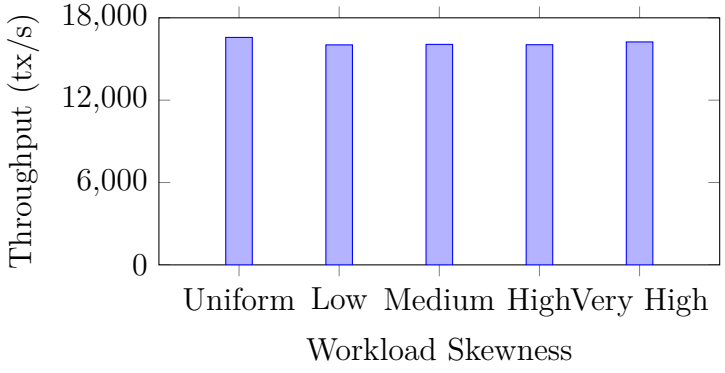


Figure 7.8: Breakdown Latency



a) With Seller Dashboard



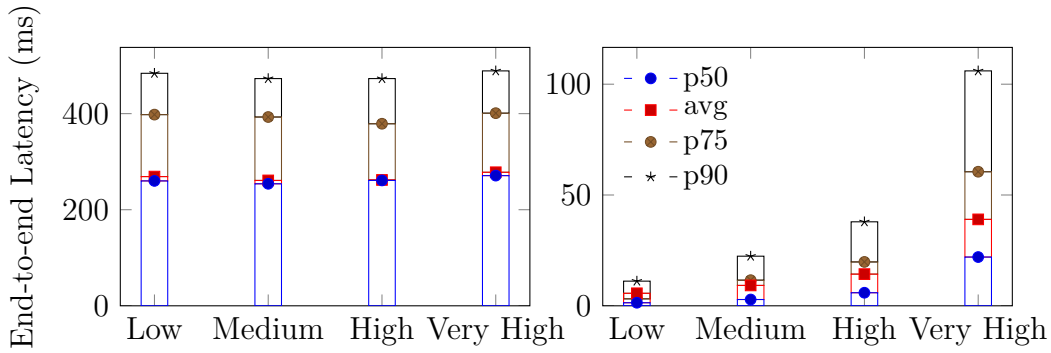
b) Without Seller Dashboard

Figure 7.9: Effect of Skew Levels on vMODB Throughput

7.6.3.3 Effect of Workload Skewness

A skewed workload leads to certain keys being accessed more frequently. In OM, a highly skewed workload leads to some sellers and their associated products being more popular than others. We follow the OM experimental procedure [146], where varying seller skewness is sufficient to stress a target system. We do not include Dapr in this experiment because there is no transactional isolation, and the concurrent data structures of the C# language drive the contention degree.

Figure 7.9 shows the effect of increasing skewness levels on throughput. For **a)**, we noticed that the increasing contention, leading to more item versions in the system and increasing memory pressure, and the high tail latency



a) Only Seller Dashboard b) Without Seller Dashboard

Figure 7.10: Effect of Skew Levels on vMODB Latency

of `Seller Dashboard`, drives the throughput decaying trend, a normal phenomenon given that very few specific sellers and their products are involved in queries. In **b)**, removing `Seller Dashboard` from the transaction ratio withdraws the decaying effect, yielding an increase of 18,29% (uniform) up to 61% (very high) in throughput. We also observe that different skewnesses have a marginal effect on throughput, demonstrating the effectiveness of vMODB concurrency primitives.

Figure 7.10 exhibits the end-to-end latency of different skew levels for Figure 7.9 **a)**. We distinguish the results of the `Seller Dashboard` (**a**) from the transactions that run in the context of a batch (**b**) for a proper analysis. We observe that `Seller Dashboard` latency shows little difference across skew levels, whereas batch-committed transactions show a typical increasing trend as skewness increases. We omit the results of vMODB run without `Seller Dashboard` because the numbers show similar no/low skewness across all levels.

7.6.3.4 Effect of Batch Size

To study the effects that the number of transactions per batch has on performance, we start by fixing the batch epoch to infinity, varying the amount of transactions in a batch, and maximizing the concurrency level. In Figure 7.11 (left), we observe an average of 1K tx/sec increase up to batch size 1K, the point where throughput stabilizes. Next, we fix the maximum num-

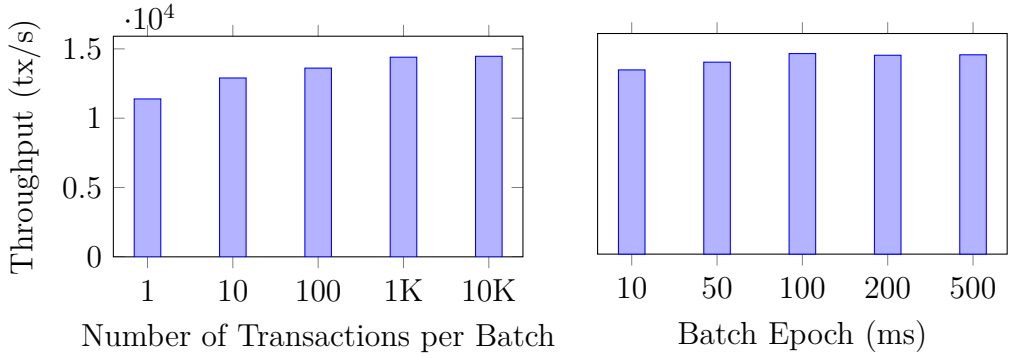


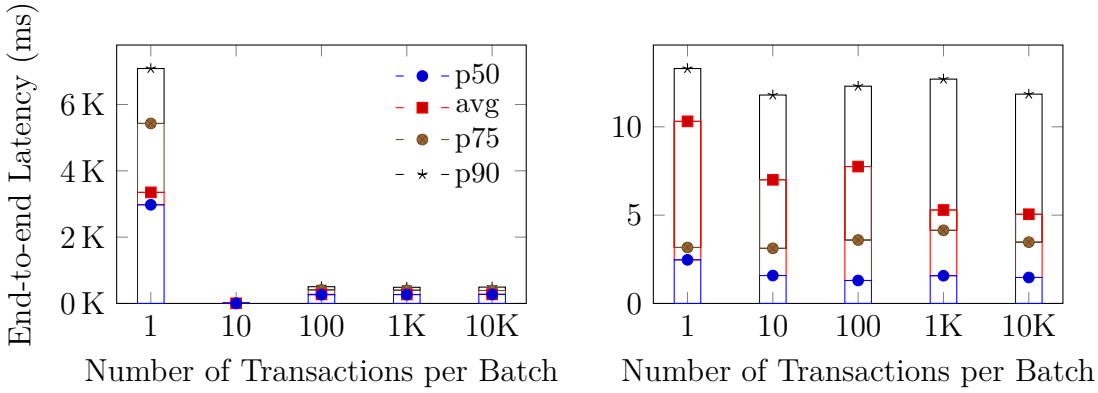
Figure 7.11: Effect of Batch Size on Throughput

ber of transactions in a batch to infinity and vary the epoch. Figure 7.11 (right) demonstrates the effectiveness of the commit protocol, showing little variation across epochs.

We then turn our attention to the effects of batch size on latency. Again, for a proper comparison of different transaction types, we separate transactions that commit within a batch from the seller dashboard query. Figure 7.12 (left) shows the significant impact of a single transaction per batch, a natural finding given the substantial amount of network I/O incurred. It is worth noting that 10 transactions per batch still leads to more than double of Dapr's throughput, providing a good trade-off to users who seek lower overall latency. Besides, Figure 7.12 (right) shows that different batch sizes lead to little variation on `Seller Dashboard` latency. On the other hand, Figure 7.13 shows that varying the batch epoch leads to a more natural variation in latency for transactions committing in a batch, without affecting throughput though.

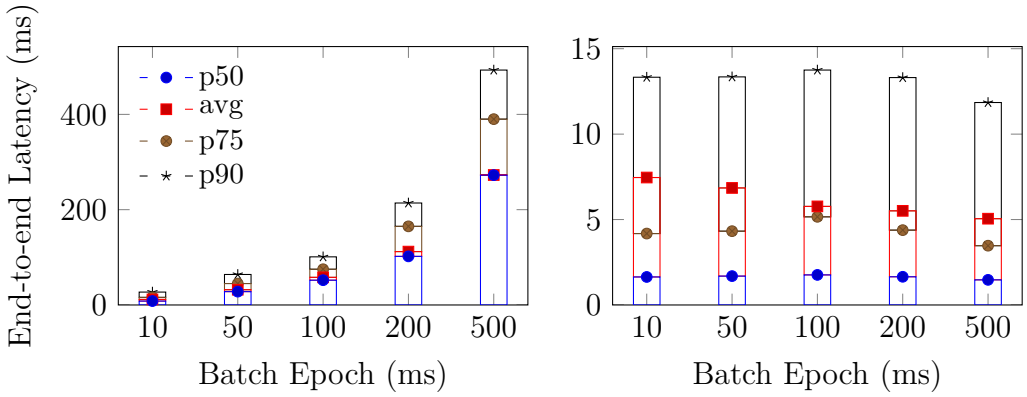
7.6.4 TPC-C

To understand the effect of scale factor and durability in TPC-C, we run vMODB with and without logging and checkpointing enabled. Figure 7.14 shows the throughput of TPC-C as the number of warehouses increases. Durability imposes an average 17% performance overhead, but this overhead reduces as the number of warehouses increases (e.g., 13% and 17% for 16 and 32 warehouses, respectively). The additional vCPUs, memory, and I/O



a) W/o Seller Dashboard b) Only Seller Dashboard

Figure 7.12: Effect of Batch Size on Latency



a) W/o Seller Dashboard b) Only Seller Dashboard

Figure 7.13: Effect of Batch Epoch on Latency

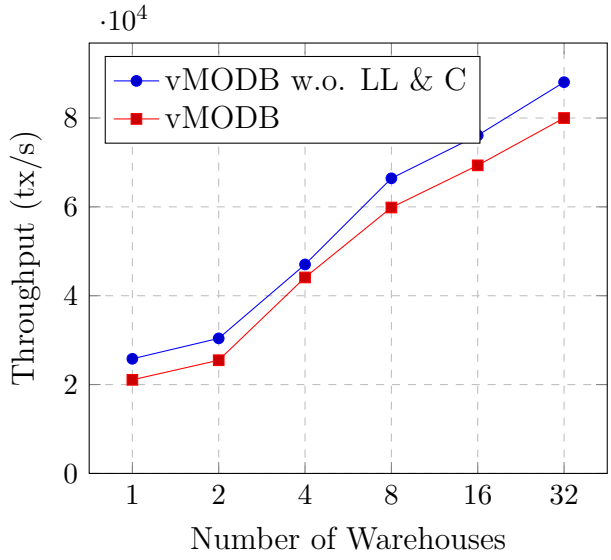


Figure 7.14: Scalability in TPC-C

speedups offered by the larger c5n instances facilitate faster durable writes, demonstrating that vMODB effectively scales with increased computational resources. In any case, vMODB shows good scalability.

Figure 7.15 reports the statistics of the intervals between the commits of each pair of adjacent batches for each scale factor in TPC-C. In vMODB, batch latencies decrease because batches are filled and processed more quickly with more requests. This effect is also observed in vMODB with logging and checkpointing, with an average overhead of 17% due to the intensive I/O incurred by the durable write procedures.

7.7 Related Work

Some database systems offer asynchronous persistent messaging through pub-/sub APIs. However, applications typically interact with DBMS via opaque libraries such as JDBC, which does not allow the execution of an ACID transaction across multiple asynchronous application components. One could port or implement EDA components as stored procedures. However, this has not emerged as a popular choice for many reasons. Moving all the business logic to stored procedures is challenging for developers because it breaks the tiered

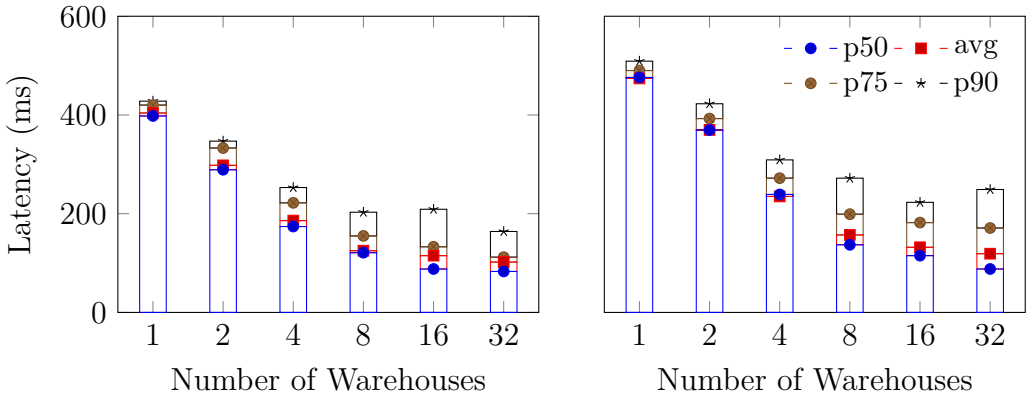


Figure 7.15: Effect of Scale Factor on Batch Latency in TPC-C. In-memory execution (left) and with durability (right)

architecture of applications and data systems and makes it cumbersome to test, verify, troubleshoot, and maintain application codes. Furthermore, having all components and associated software artifacts managed and executed within the DBMS can jeopardize the key EDA principles, such as fault isolation of components, loose coupling, independent deployment and schema changes, and fine-grained scalability [133, 204].

Function as a Service (FaaS) and dataflow systems such as Statefun [239], Netherite [31], and Portals [229] offer stronger execution guarantees, however, as Dapr, they do not offer multi-component ACID. Similarly to Orleans [34] and Snapper [157], all offer programming models that force partitioning the application into fine-grained objects with single-thread access following the actor model [3] and lack relational data model, data constraints, and query processing, which can be restrictive for a broad range of data-intensive applications and developers [190].

Distributed frameworks such as Ambrosia [104] and Darq [152] offer strong execution guarantees to application components but are oblivious to data management and transactional guarantees.

7.8 Conclusion

Distributed applications that employ the popular EDA often face data consistency and integrity challenges. vMODB is a distributed framework that facilitates the development of EDAs with high data consistency and integrity. vMODB's VMS programming model offers familiar constructs that enable transparent ACID transactions and data constraint enforcement across multiple components while withholding the advantages of EDA for developers. Based on the application semantics captured from the VMS model, vMODB's coordinator service teams up with the per-VMS multi-version database to schedule and execute transactions deterministically and efficiently. Our evaluation using two benchmarks demonstrates that vMODB significantly outperforms Dapr, a widely adopted distributed framework that only achieves eventual consistency.

Acknowledgments

We thank Marcos Antonio Vaz Salles for contributions during the early versions of this work.

Chapter 8

Concluding Remarks

From a high-level perspective, this thesis addresses a long-standing open question: Are event-driven architectures necessarily fated to the eventual consistency model?

The popularity of high-level, object-oriented languages with performant application runtimes and APIs that abstract away database tasks such as schema, index, transaction, and connection management, including facilities that decrease the friction of object-relational impedance mismatch [18], called the attention of database researchers to the fact that stored procedures were no longer the de facto programming abstraction and serializable isolation was not the first choice in data-intensive applications [199, 249].

The advent of the cloud just exacerbated the distance of application programmers from databases. By using microservices, a widely adopted distributed application architectural style in the cloud, as a reference, our large-scale study brings fresh perspectives to the database community through a thorough analysis of how modern distributed applications interact with database systems. In particular, we found that microservice architectures are composed of functionally partitioned applications; that is, functionalities and the associated data they operate on are partitioned into smaller, independent services, largely adopting a model of exchanging data and communicating operations asynchronously via events.

The key insight is that a significant amount of data flows outside the database [112], relegating databases to durable storage for the many components that perform operations and manage data at the application level. Furthermore, it revealed additional underlying assumptions from developers that were unclear to the database research community. Practitioners pursue

a decentralized computing and data management model not only to reap performance benefits but also to better adhere to modern software engineering practices that are facilitated by the cloud. In particular, software teams seek to implement, deploy, scale, specialize, upgrade, and fix software faster and independently. Microservice principles of fine-grained components facilitate this in the cloud, e.g., through containerized deployments.

Although this results in transactions no longer being held in a single component but rather traversing several heterogeneous components that are often integrated through message middleware, relegated to an eventual consistency model, the paradigm shift supports the organic growth of companies, allowing team separation and adaptable resource and scalability demands for each. We believe that, rather than prompting practitioners to move their applications back to a monolithic design if data integrity is important, database systems designers must find alternatives to better serve database users' new needs.

A natural deduction from these insights was that existing state-of-the-art benchmarks could not measure the data management overhead and complexity of this emerging class of applications. This thesis then proposes Online Marketplace, a novel microservice benchmark that closes the benchmarking gap by allowing the comparison and evaluation of data platforms for microservice-based applications and other trending cloud application architectures and programming models, including modular monolithic architectures, FaaS, and Actors. As a result, the Online Marketplace benchmark is well-positioned to serve as a testbed for evaluating and comparing future data systems for cloud-native applications.

Finally, equipped with a corpus of requirements encompassing data management and software engineering practices sought in practice, together with appropriate benchmarking, this thesis proposes a novel database architecture to address the challenges of modern data-intensive applications in the cloud. We rethink traditional DBMS architecture and propose a cross-stack design to tame the "data on the outside" conundrum. Virtual microservices, a novel programming model that packages together data, application logic, concurrency semantics, and component data dependencies, are leveraged as database abstractions, enabling minimized coordination for highly scalable distributed transactions while maintaining the benefits of event-driven and microservice architectures in the cloud. Contrasting with existing wisdom, we demonstrate that event-driven and microservice architectures do not necessarily need to give up strong transactional guarantees. As a result, vMODB

is properly positioned to be a strong competitor in the cloud landscape in the coming years.

Overall, this thesis highlights the importance of engaging with database users to understand and characterize their practice, identify new research opportunities, and advance data management for classes of applications that are overlooked by the database research community.

8.1 Ongoing and Future Work

The contributions of this thesis have motivated and spawned additional research lines. In this section, we outline ongoing and promising directions for future research.

Generalizability of Virtual Micro Services. One of the main limitations of state-of-the-art programming models for the cloud consists of forcing application programmers to port stateless and procedural programs to a model where applications are partitioned into objects that encapsulate opaque objects as states with single-thread access. Despite the apparent simplified programmability, key data management tasks are jeopardized in this model. For instance, operations on a large number of data items would involve a substantial number of objects, often distributed across nodes in a cluster, resulting in high coordination costs [190].

Virtual microservices address this gap by providing a general programming model that aligns with how data-intensive applications traditionally interact with database systems. They maintain the attractiveness of a stateless design and transparently offer ACID guarantees with a rich data model.

This generality can be further explored to map other programming models to take advantage of vMODB support. The use of Online Marketplace specifications to advance data management in cloud-native actor systems [156] suggests that the concurrency primitives in vMODB can be leveraged to express actor-like semantics. Similarly, FaaS applications could also be ported to virtual microservices without losing functionality. The design of hybrid-paradigm applications can motivate additional performance strategies for application builders.

Accelerating the vMODB data path. With the effectiveness of the coordination minimization strategies employed by vMODB, particularly in the VMS transaction execution scope, we anticipate that the data processing costs will remain major in networking. For instance, 88% of data transfer

costs in event-driven architectures come from the network [107]. As VMSes are designed for distributed settings, it is paramount to investigate techniques to prevent networking from becoming the bottleneck.

Preliminary experiments adapting vMODB network modules to use the scatter/gather I/O offered by *epoll* in Linux [159] have provided up to 40% performance improvements. Despite recent advances in high-performance APIs, such as asynchronous I/O with io-uring, and kernel-bypass techniques, such as demikernels [281], it is unclear how to effectively leverage these advances in runtime applications, as supported by vMODB.

Serverless vMODB. We envision a cloud computing service model that offers virtual microservices as a service (vMaaS). In this model, developers enjoy all the benefits of vMODB while relegating the deployment, monitoring, upgrading, scaling, and failure handling of virtual microservice components to the vMaaS platform. This project promises to span different venues of investigation, including how to run virtual microservices with high performance and decreased carbon footprint, how to decrease downtime in the presence of network partitions and crashes while mitigating a high data replication cost, how to isolate resources efficiently across tenants, and how to allow developers to tune their virtual microservices to different consistency levels to meet performance constraints.

Advancing Online Marketplace. The *Online Marketplace* benchmark makes a principled contribution to the database community. By focusing on real-world requirements and challenges, it dissects the key data management workloads and tasks practitioners struggle with. To further foment adoption and tackle additional challenges, we envision at least three advancements:

(i) Portability: An interesting line is the creation of libraries in different programming languages and models to facilitate and speed up the implementation of the application prescribed by *Online Marketplace* on different data platforms. Such a contribution promises to reduce the initial adoption barrier and define a reliable basis for comparing emerging competing data platforms.

(ii) Dynamic workloads: An important property of microservice architectures and general cloud applications is adapting resource usage according to online demand. The *Online Marketplace* can incorporate new criteria related to scaling and stepping down resources holistically, that is, according to the workload each microservice faces. This advancement can strengthen the usefulness of the *Online Marketplace* benchmark for cloud-native deployments.

(iii) Fault tolerance: Evaluating fault tolerance is also a key advance-

ment since it would allow the *Online Marketplace*benchmark to measure the state-of-the-art data platforms' performance under failure. The *Online Marketplace*'s driver can simulate crashes in specific components, nodes, and network failures and delays. To assist this process, the *Online Marketplace* and the target platform under evaluation can benefit from containerized infrastructure.

Monitoring data invariants. An important line of investigation lies in devising effective and efficient monitoring tools to assist application developers in identifying data integrity violations. We anticipate that some components of an application may remain black boxes to the database due to, for example, data privacy concerns [253], proprietary technology, and legacy systems, preventing the realization of the benefits of vMODB.

In this vein, this thesis has motivated developments in two work streams: (i) online monitoring of data integrity violations [63] and (ii) event-based constraint enforcement [151]. The first builds on the insight that, because event payloads exchanged by components contain semantic information about data and operations that ought to be applied to application states, they can be leveraged as appropriate abstractions to identify violations of data integrity properties in a non-intrusive manner. The second posits that, in the absence of application control, the definition of event streams can be augmented with application semantics in the form of invariants. The invariants are then used to prevent processing event streams that could lead to data integrity violations, for example, by enforcing a safe event-processing order. Both work streams are under active development.

Bibliography

[1] martinothamar (Github User). *InconsistentStateException with ADO.NET SQL Server grain storage*. 2018. URL: https://github.com/dotnet/orleans/issues/4697.

[2] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-oriented programming". In: *ECOOP'97 — Object-Oriented Programming*. 1997, pp. 220–242.

[3] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. en. The MIT Press, 1986. ISBN: 978-0-262-25555-4. DOI: 10.7551/mitpress/1086.001.0001. URL: https://direct.mit.edu/books/book/4794/ActorsA-Model-of-Concurrent-Computation-in (visited on 05/29/2024).

[4] Akka. *Build and run apps that react to change*. 2024. URL: https://akka.io.

[5] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. "Boom analytics: exploring data-centric, declarative programming for the cloud". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: Association for Computing Machinery, 2010, pp. 223–236. ISBN: 9781605585772. DOI: 10.1145/1755913.1755937. URL: https://dl.acm.org/doi/10.1145/1755913.1755937.

[6] Andreas Andreakis, Falguni Jhaveri, Ioannis Papapanagiotou, Mark Cho, Poorna Reddy, and Tongliang Liu. *Delta: A Data Synchronization and Enrichment Platform*. Netflix, 2019. URL: https://netflixtechblog.com/delta-a-data-synchronization-and-enrichment-platform-e82c36a79aee.

[7] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. "Linear Road: A Stream Data Management Benchmark". In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. Toronto, Canada: VLDB Endowment, 2004, pp. 480–491. ISBN: 0120884690.

[8] GitHub user arielmoraes. *#700*. 2018. URL: https://github.com/dotnet-architecture/eShopOnContainers/issues/700.

[9] Apache Avro. *Apache Avro: a data serialization system*. (Accessed on 28/11/2024). URL: https://avro.apache.org.

[10] AWS. *Amazon EC2 C5 Instances*. 2024. URL: https://aws.amazon.com/ec2/instance-types/c5/.

[11] AWS. *Lambda*. (Accessed on 27/11/2024). URL: https://aws.amazon.com/lambda/.

[12] AWS. *What is EDA (Event-Driven Architecture)?* 2025. URL: https://aws.amazon.com/what-is/eda/.

[13] Amazon AWS. *Microservices*. Amazon, 2024. URL: https://aws.amazon.com/microservices.

[14] Leonardo Guerreiro Azevedo, Rodrigo da Silva Ferreira, Viviane Torres da Silva, Maximillien de Bayser, Elton F. de S. Soares, and Raphael Melo Thiago. "Geological Data Access on a Polyglot Database Using a Service Architecture". In: *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*. SBCARS '19. Salvador, Brazil: ACM, 2019, pp. 103–112. ISBN: 978-1-4503-7637-2. DOI: 10.1145/3357141.3357603. URL: http://doi.acm.org/10.1145/3357141.3357603.

[15] Microsoft Azure. *A technical overview of Azure Cosmos DB*. Microsoft. URL: https://azure.microsoft.com/en-us/blog/a-technical-overview-of-azure-cosmos-db.

[16] Microsoft Azure. *Microservice Applications*. Microsoft, 2024. URL: https://azure.microsoft.com/en-us/solutions/microservice-applications.

[17] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. "Highly available transactions: Virtues and limitations". In: *Proceedings of the VLDB Endowment* 7.3 (2013), pp. 181–192.

[18] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. "Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1327–1342. ISBN: 9781450327589. DOI: 10.1145/2723372.2737784. URL: https://doi.org/10.1145/2723372.2737784.

[19] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. "Scalable atomic visibility with RAMP transactions". In: *ACM Transactions on Database Systems (TODS)* 41.3 (2016), pp. 1–45.

[20] Peter Bailis and Ali Ghodsi. "Eventual Consistency Today: Limitations, Extensions, and Beyond". In: *Queue* 11.3 (Mar. 2013), pp. 20–32. ISSN: 1542-7730. DOI: 10.1145/2460276.2462076. URL: https://doi.org/10.1145/2460276.2462076.

[21] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. "Orca: A language for parallel programming of distributed systems". In: *IEEE transactions on software engineering* 18.March (1992), pp. 190–205.

[22] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. "Programming languages for distributed computing systems". In: *ACM Comput. Surv.* 21.3 (Sept. 1989), pp. 261–322. ISSN: 0360-0300. DOI: 10.1145/72551.72552. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/72551.72552.

[23] Kenny Bastani. *event-stream-processing-microservices*. URL: https://github.com/kbastani/event-stream-processing-microservices.

[24] Adam Bellemare. *Building Event-Driven Microservices: Leveraging Organizational Data at Scale*. Sebastopol, CA: O'Reilly Media, 2020. ISBN: 9781492057895.

[25] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. *Orleans: Distributed Virtual Actors for Programmability and Scalability*. Tech. rep. MSR-TR-2014-41. Microsoft, Mar. 2014. URL: https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/.

[26] Philip A. Bernstein. "Resurrecting Middle-Tier Distributed Transactions". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 42.2 (June 2019), pp. 3–6.

[27] Philip A. Bernstein and Sergey Bykov. "Developing Cloud Services Using the Orleans Virtual Actor Model". In: *IEEE Internet Computing* 20.5 (2016), pp. 71–75.

[28] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[29] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009. ISBN: 1558606238.

[30] BroadleafCommerce. *Broadleaf Microservices Architecture*. Broadleaf Commerce. URL: https://www.broadleafcommerce.com/product/microservice-architecture.

[31] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. "Netherite: efficient execution of serverless workflows". In: *Proc. VLDB Endow.* 15.8 (Apr. 2022), pp. 1591–1604. ISSN: 2150-8097. DOI: 10.14778/3529337.3529344. URL: https://doi.org/10.14778/3529337.3529344.

[32] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. "Durable functions: semantics for stateful serverless". In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485510. URL: https://doi.org/10.1145/3485510.

[33] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. "Orleans: cloud computing for everyone". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, Oct. 2011, pp. 1–14. ISBN: 978-1-4503-0976-9. DOI: 10.1145/2038916.2038932. (Visited on 11/24/2023).

[34] Sergey Bykov, Alan Geller, Gabriel Kliot, Jim Larus, Ravi Pandya, and Jorgen Thelin. *Orleans: A Framework for Cloud Computing*. Tech. rep. MSR-TR-2010-159. Microsoft, Nov. 2010. URL: https:

//www.microsoft.com/en-us/research/publication/orleans-a-framework-for-cloud-computing/.

[35] Phil Calçado. *The Back-end for Front-end Pattern (BFF)*. 2015. URL: https://philcalcado.com/2015/09/18/the%5C_back%5C_end%5C_for%5C_front%5C_end%5C_pattern%5C_bff.html.

[36] Martin Campbell-Kelly. *From Airline Reservations to Sonic the Hedgehog: A history of the software industry*. The MIT Press, Cambridge Massachusetts Software Markets newsletters, Highdon Publishing, London, 2003.

[37] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink™: Stream and Batch Processing in a Single Engine". English. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015), pp. 28–38.

[38] Andrés Carrasco, Brent van Bladel, and Serge Demeyer. "Migrating towards Microservices: Migration and Architecture Smells". In: *Proceedings of the 2nd International Workshop on Refactoring*. IWoR 2018. Montpellier, France: Association for Computing Machinery, 2018, pp. 1–6. ISBN: 9781450359740. DOI: 10.1145/3242163.3242164.

[39] Luiz Carvalho, Alessandro Garcia, Wesley KG Assunção, Rafael de Mello, and Maria Julia de Lima. "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. 2019, pp. 22–29.

[40] Marcus Cavalcanti. *Lessons learned about running Microservices*. B2W, 2020. URL: https://medium.com/b2w-engineering-en/lessons-learned-about-running-microservices-a51d952cb50b.

[41] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Trans. Comput. Syst.* 3.1 (1985), pp. 63–75.

[42] Brian Chavez. *Bogus*. Bogus is fundamentally a C# port of faker.js and inspired by FluentValidation's syntax sugar. Microsoft, 2024. URL: https://github.com/bchavez/Bogus.

[43] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. "Detecting Performance Anti-Patterns for Applications Developed Using Object-Relational Mapping". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 1001–1012. ISBN: 9781450327565. DOI: 10.1145/2568225.2568259. URL: https://doi.org/10.1145/2568225.2568259.

[44] Chaoyi Cheng, Mingzhe Han, Nuo Xu, Spyros Blanas, Michael D Bond, and Yang Wang. "Developer's Responsibility or Database's Responsibility? Rethinking Concurrency Control in Databases". In: *13th Annual Conference on Innovative Data Systems Research (CIDR'23). January 8-11, 2023, Amsterdam, The Netherlands.* 2023.

[45] Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Mae Milano. "New Directions in Cloud Programming". In: *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. URL: http://cidrdb.org/cidr2021/papers/cidr2021%5C_paper16.pdf.

[46] Jon Chew. *Avoiding Double Payments in a Distributed Payments System*. AirBnb, 2019. URL: https://medium.com/airbnb-engineering/avoiding-double-payments-in-a-distributed-payments-system-2981f6b070bb.

[47] Dmitry Chornyi. *Engineering Uber's Next-Gen Payments Platform*. Uber Technologies Inc., 2018. URL: https://eng.uber.com/payments-platform.

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

[49] Alibaba Cloud. *Microservices Engine*. Alibaba, 2024. URL: https://www.alibabacloud.com/en/product/microservices-engine.

[50] Google Cloud. *What is Microservices Architecture?* Google, 2024. URL: https://cloud.google.com/learn/what-is-microservices-architecture.

[51] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. "The mixed workload CH-benCHmark". In: *Proceedings of the Fourth International Workshop on Testing Database Systems*. DBTest '11. Athens, Greece: Association for Computing Machinery, 2011. ISBN: 9781450306553. DOI: 10.1145/1988842.1988850. URL: https://doi.org/10.1145/1988842.1988850.

[52] Dan Conger. *How Apache Kafka Enables Podium to "Ship It and See What Happens"*. Confluent, 2021. URL: https://www.confluent.io/blog/how-apache-kafka-enables-podium-to-ship-it-and-see-what-happens.

[53] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152. URL: https://doi.org/10.1145/1807128.1807152.

[54] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. "Spanner: Google's Globally Distributed Database". In: *ACM Transactions on Computer Systems* 31.3 (Aug. 2013). ISSN: 0734-2071. DOI: 10.1145/2491245. URL: https://doi.org/10.1145/2491245.

[55] TPC Council. *TPC Benchmark C revision*. Revision 5.11. 2010. URL: https://www.tpc.org/tpcc.

[56] TPC Council. *TPC-W: Benchmarking An Ecommerce Solution*. Revision 1.2. 2005. URL: https://www.tpc.org/tpcw/tpc-w_wh.pdf.

[57] P. Cruz, H. Astudillo, R. Hilliard, and M. Collado. "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)*. 2019, pp. 174–181.

[58] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. "The snowflake elastic data warehouse". In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 215–226.

[59] Dapr. *Dapr is a portable, event-driven, runtime for building distributed applications across cloud and edge*. 2024. URL: https://github.com/dapr/dapr.

[60] Dapr. *Dapr SDK for .NET*. URL: https://github.com/dapr/dotnet-sdk.

[61] Dapr. *Dependency graph – Dapr SDK for .NET*. URL: https://github.com/dapr/dotnet-sdk/network/dependents.

[62] Jérôme Darmont and Michel Schneider. "Object-Oriented Database Benchmarks". In: *Advanced Topics in Database Research, Vol. 1*. Ed. by Keng Siau. Idea Group, 2002, pp. 34–57.

[63] Prangshuman Das, Rodrigo Laigner, and Yongluan Zhou. "HawkEDA: a tool for quantifying data integrity violations in event-driven microservices". In: *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*. DEBS '21. Virtual Event, Italy: Association for Computing Machinery, 2021, pp. 176–179. ISBN: 9781450385558. DOI: 10.1145/3465480.3467838. URL: https://doi.org/10.1145/3465480.3467838.

[64] Datadog. *What Is Audit Logging?* URL: https://www.datadoghq.com/knowledge-center/audit-logging.

[65] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. "Transactions across serverless functions leveraging stateful dataflows". In: *Information Systems* 108 (2022), p. 102015. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2022.102015. URL: https://www.sciencedirect.com/science/article/pii/S0306437922000229.

[66] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: Amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[67] Dapr Docs. *Redis Streams*. URL: https://docs.dapr.io/reference/components-reference/supported-pubsub/setup-redis-pubsub/.

[68] docker docs. *Run multiple services in a container*. (Accessed on 2021-03-15). 2021. URL: https://docs.docker.com/config/containers/multi-service_container.

[69] Microsoft Orleans documentation. *Grain interface versioning*. (Accessed on 07/10/2024). URL: https://learn.microsoft.com/en-us/dotnet/orleans/grains/grain-versioning/grain-versioning.

[70] PostgreSQL 12 Documentation. *NOTIFY*. URL: https://www.postgresql.org/docs/current/sql-notify.html.

[71] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. "RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications". In: *ACM Trans. Storage* 17.4 (Oct. 2021). ISSN: 1553-3077. DOI: 10.1145/3483840. URL: https://doi.org/10.1145/3483840.

[72] DuckDB. *DuckDB*. 2024. URL: https://github.com/duckdb/duckdb.

[73] Tamer Eldeeb, Sebastian Burckhardt, Reuben Bond, Asaf Cidon, Junfeng Yang, and Philip A. Bernstein. "Cloud Actor-Oriented Database Transactions in Orleans". In: *Proc. VLDB Endow.* 17.12 (2024), pp. 3720–3730. URL: https://www.vldb.org/pvldb/vol17/p3720-eldeeb.pdf.

[74] B2W Engineering. *restQL: Tackling microservice query complexity*. (Accessed on 2021-02-28). 2018. URL: https://medium.com/b2w-engineering-en/restql-tackling-microservice-query-complexity-27def5d09b40.

[75] Uber Engineering. *Revolutionizing Money Movements at Scale with Strong Data Consistency*. 2020. URL: https://eng.uber.com/money-scale-strong-data.

[76] eShopOnContainers. *.NET Application Architecture - Reference Apps*. URL: https://github.com/dotnet-architecture/eShopOnContainers.

[77] João Esteves, Rosa Costa, Yongluan Zhou, and Ana Almeida. "An exploratory analysis of methods for real-time data deduplication in streaming processes". In: *Proceedings of the 17th ACM International Conference on Distributed and Event-Based Systems*. DEBS '23. Neuchatel, Switzerland: Association for Computing Machinery, 2023, pp. 91–102. ISBN: 9798400701221. DOI: 10.1145/3583678.3596898. URL: https://doi.org/10.1145/3583678.3596898.

[78] Stephan Ewen. *Stateful Functions 2.0 - An Event-driven Database on Apache Flink*. 2020. URL: https://flink.apache.org/2020/04/07/stateful-functions-2.0-an-event-driven-database-on-apache-flink.

[79] Azure Service Fabric. *Service Fabric programming model overview*. URL: https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-choose-framework.

[80] Rafael Ferreira. *Microservices at Nubank, an overview*. Nubank, 2019. URL: https://building.nubank.com.br/microservices-at-nubank-an-overview.

[81] Rafael Ferreira and Edward Wible. *Architecting a Modern Financial Institution*. Nubank, 2017. URL: https://www.infoq.com/presentations/nubank-architecture.

[82] João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. "Antipode: Enforcing Cross-Service Causal Consistency in Distributed Applications". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. Koblenz, Germany: Association for Computing Machinery, 2023, pp. 298–313. ISBN: 9798400702297. DOI: 10.1145/3600006.3613176. URL: https://doi.org/10.1145/3600006.3613176.

[83] Flask. *Python Flask*. (Accessed on 16/12/2024). URL: https://flask.palletsprojects.com/en/stable/.

[84] Flink. *Upgrading Applications and Flink Versions*. URL: https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/ops/upgrading/.

[85] Apache Flink. *Flink Architecture*. Apache Flink. URL: https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture.

[86] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201485672.

[87] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. URL: http://%20martinfowler.com/articles/injection.html (visited on 01/23/2004).

[88] Martin Fowler. *InversionOfControl*. (Accessed on 2021-03-19). 2005. URL: https://martinfowler.com/bliki/InversionOfControl.html.

[89] Martin Fowler. *Microservices a definition of this new architectural term*. 2014. URL: https://martinfowler.com/articles/microservices.html.

[90] Martin Fowler. *PolyglotPersistence*. 2011. URL: https://martinfowler.com/bliki/PolyglotPersistence.html.

[91] Martin Fowler. *Repository*. URL: https://martinfowler.com/eaaCatalog/repository.html.

[92] Martin Fowler. *Service Layer*. URL: https://martinfowler.com/eaaCatalog/serviceLayer.html.

[93] Martin Fowler. *What do you mean by "Event-Driven"?* 2017. URL: https://martinfowler.com/articles/201701-event-driven.html.

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

[95] Spring Frawework. *16. Transaction Management*. (Accessed on 2021-03-02). 2021. URL: https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/transaction.html.

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

[97] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. "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*. 2019, pp. 3–18.

[98] Hector Garcia-Molina and Kenneth Salem. "Sagas". In: *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. SIGMOD '87. San Francisco, California, USA: Association for Computing Machinery, 1987, pp. 249–259. ISBN: 0897912365. DOI: 10.1145/38713.38742. URL: https://doi.org/10.1145/38713.38742.

[99] Johannes Gehrke and Raghu Ramakrishnan. *Database management systems*. McGraw-Hill, 2003.

[100] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B. Kahveci, Ali Gürbüz, Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, Ufuk Yılmaz, Mehmet Doğan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. "Hazelcast jet: low-latency stream processing at the 99.99th percentile". In: *Proc. VLDB Endow.* 14.12 (July 2021), pp. 3110–3121. ISSN: 2150-8097. DOI: 10.14778/3476311.3476387. URL: https://doi.org/10.14778/3476311.3476387.

[101] Javad Ghofrani and Daniel Lübke. "Challenges of Microservices Architecture: A Survey on the State of the Practice". In: *Proceedings of the 10th Central European Workshop on Services and their Composition, Dresden, Germany, February 8-9, 2018*. Ed. by Nico Herzberg, Christoph Hochreiner, Oliver Kopp, and Jörg Lenhard. Vol. 2072. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 1–8.

[102] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

[103] Adam Gluck. *Introducing Domain-Oriented Microservice Architecture*. Uber Technologies Inc., 2022. URL: https://www.uber.com/en-DK/blog/microservice-architecture.

[104] Jonathan Goldstein, Ahmed Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Ghosh Peshawaria, Tal Zaccai, and Irene Zhang. "A.M.B.R.O.S.I.A: providing performant virtual resiliency for distributed applications". In: *Proc. VLDB Endow.* 13.5 (Jan. 2020), pp. 588–601. ISSN: 2150-8097. DOI: 10.14778/3377369.3377370. URL: https://doi.org/10.14778/3377369.3377370.

[105] Jean-Philippe Gouigoux and Dalila Tamzalit. "From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 62–65. DOI: 10.1109/ICSAW.2017.35.

[106] G. Graefe and W.J. McKenna. "The Volcano optimizer generator: extensibility and efficient search". In: *Proceedings of IEEE 9th International Conference on Data Engineering*. 1993, pp. 209–218. DOI: 10.1109/ICDE.1993.344061.

[107] Rachel Groberman. *Unlock Cost Savings with Freight Clusters–Now in General Availability*. URL: https://www.confluent.io/blog/freight-clusters-are-generally-available.

[108] Network Working Group. *The Idempotency HTTP Header Field*. 2020. URL: https://datatracker.ietf.org/doc/html/draft-idempotency-header-00.

[109] Christopher Gustafson. "Improving Availability of Stateful Serverless Functions in Apache Flink". Master thesis. KTH Royal Institute of Technology, 2022.

[110] James Hamilton. "Keynote talk". In: *The 3rd ACM SIGOPS International Workshop on Large-Scale Distributed Systems and Middleware (LADIS '09)*. 2009.

[111] Wilhelm Hasselbring and Guido Steinacker. "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 243–246. DOI: 10.1109/ICSAW.2017.11.

[112] Pat Helland. "Data on the outside versus data on the inside". In: *Commun. ACM* 63.11 (Oct. 2020), pp. 111–118. ISSN: 0001-0782. DOI: 10.1145/3410623. URL: https://doi.org/10.1145/3410623.

[113] Pat Helland. "Immutability Changes Everything". In: *Commun. ACM* 59.1 (2015), pp. 64–70.

[114] Pat Helland. "Life beyond distributed transactions". In: *Commun. ACM* 60.2 (Jan. 2017), pp. 46–54. ISSN: 0001-0782. DOI: 10.1145/3009826. URL: https://doi.org/10.1145/3009826.

[115] Joseph M. Hellerstein and Peter Alvaro. "Keeping CALM: When Distributed Consistency is Easy". In: *Communications of the ACM* 63.9 (Aug. 2020), pp. 72–81. ISSN: 0001-0782. DOI: 10.1145/3369736.

[116] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. *Serverless Computing: One Step Forward, Two Steps Back*. 2018. arXiv: 1812.03651 [cs.DC].

[117] User: hendoe. *Approaches to update microservices databases retroactively*. (Accessed on 07/10/2024). URL: https://stackoverflow.com/questions/59923240/approaches-to-update-microservices-databases-retroactively.

[118] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. "Distributed transactions on serverless stateful functions". In: *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*. 2021, pp. 31–42.

[119] Hibernate. *Hibernate ORM*. URL: https://hibernate.org/orm/.

[120] Gregor Hohpe. "Let's have a conversation". In: *IEEE internet computing* 11.3 (2007), pp. 78–81.

[121] Gregor Hohpe and Bobby Woolf. *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*. Boston, MA: Addison-Wesley Professional, 2004. ISBN: 0321322118.

[122] Gansen Hu, Zhaoguo Wang, Chuzhe Tang, Jiahuan Shen, Zhiyuan Dong, Sheng Yao, and Haibo Chen. "WeBridge: Synthesizing Stored Procedures for Large-Scale Real-World Web Applications". In: *Proc. ACM Manag. Data* 2.1 (Mar. 2024). DOI: 10.1145/3639319. URL: https://doi.org/10.1145/3639319.

[123] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. "Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows". In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, July 2023, pp. 419–432. ISBN: 978-1-939133-35-9. URL: https://www.usenix.org/conference/atc23/presentation/huye.

[124] IATA. *Airline Retailing*. IATA. URL: https://www.iata.org/en/programs/airline-distribution/retailing.

[125] IBM. *Architectural characteristics of web-based applications*. https://www.ibm.com/docs/en/db2-for-zos/12?topic=environment-architectural-characteristics-web-based-applications. 2024.

[126] imbursepayments. *What is a PSP and how does it work?* URL: https://imbursepayments.com/what-is-a-psp.

[127] Materialize Inc. *Materialize: The Operational Data Warehouse*. URL: https://materialize.com.

[128] Kasun Indrasiri and Danesh Kuruppu. *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020.

[129] Zhipeng Jia and Emmett Witchel. "Boki: Stateful Serverless Computing with Shared Logs". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 691–707. ISBN: 9781450387095. DOI: 10.1145/3477132.3483541. URL: https://doi.org/10.1145/3477132.3483541.

[130] Apache Kafka. *Kafka Streams*. URL: https://kafka.apache.org/27/documentation/streams.

[131] Asterios Katsifodimos and Marios Fragkoulis. "Operational Stream Processing: Towards Scalable and Consistent Event-Driven Applications". In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. OpenProceedings.org, 2019, pp. 682–685. DOI: 10.5441/002/edbt.2019.86. URL: https://doi.org/10.5441/002/edbt.2019.86.

[132] Ninad Khisti. *Measuring Transactional Integrity in Airbnb's Distributed Payment Ecosystem*. AirBnb, 2018. URL: https://medium.com/airbnb-engineering/measuring-transactional-integrity-in-airbnbs-distributed-payment-ecosystem-a670d6926d22.

[133] Martin Kleppmann, Alastair R. Beresford, and Boerge Svingen. "Online Event Processing: Achieving Consistency Where Distributed Transactions Have Failed". In: *Queue* 17.1 (2019), pp. 116–136.

[134] Holger Knoche and Wilhelm Hasselbring. "Drivers and Barriers for Microservice Adoption - A Survey among Professionals in Germany". In: *Enterprise Modelling and Information Systems Architectures (EMISAJ)* 14 (Jan. 2019), pp. 1–35. DOI: 10.18417/emisa.14.1.

[135] Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Matei Zaharia, and Xiangyao Yu. "Epoxy: ACID Transactions across Diverse Data Stores". In: *Proc. VLDB Endow.* 16.11 (July 2023), pp. 2742–2754. ISSN: 2150-8097. DOI: 10.14778/3611479.3611484. URL: https://doi.org/10.14778/3611479.3611484.

[136] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. "The Case for Learned Index Structures". In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 489–504.

[137] Jay Kreps, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing". In: *Proceedings of the NetDB*. Vol. 11. Athens, Greece. 2011, pp. 1–7.

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

[139] Hsiang-Tsung Kung and John T Robinson. "On optimistic methods for concurrency control". In: *ACM Transactions on Database Systems (TODS)* 6.2 (1981), pp. 213–226.

[140] Rodrigo Laigner. *EventBenchmark*. 2023. URL: https://github.com/diku-dk/EventBenchmark/releases/tag/v1.0.

[141] Rodrigo Laigner, Ana Carolina Almeida, Wesley K. G. Assunção, and Yongluan Zhou. "An Empirical Study on Challenges of Event Management in Microservice Architectures". In: *ACM Trans. Softw. Eng. Methodol.* (Dec. 2025). Just Accepted. ISSN: 1049-331X. DOI: 10.1145/3776581. URL: https://doi.org/10.1145/3776581.

[142] Rodrigo Laigner, George Christodoulou, Kyriakos Psarakis, Asterios Katsifodimos, and Yongluan Zhou. "Transactional Cloud Applications: Status Quo, Challenges, and Opportunities". In: *Proceedings of the 2025 International Conference on Management of Data*. SIGMOD '25. Berlim, Germany: Association for Computing Machinery, 2025.

[143] Rodrigo Laigner, Marcos Kalinowski, Pedro Diniz, Leonardo Barros, Carlos Cassino, Melissa Lemos, Darlan Arruda, Sergio Lifschitz, and Yongluan Zhou. "From a Monolithic Big Data System to a Microservices Event-Driven Architecture". In: *46th Euromicro Conference on Software Engineering and Advanced Applications*. 2020, pp. 213–220.

[144] Rodrigo Laigner, Sérgio Lifschitz, Marcos Kalinowski, Marcus Poggi, and Marcos Antonio Vaz Salles. "Towards a Technique for Extracting Relational Actors from Monolithic Applications". In: *Anais do XXXIV Simpósio Brasileiro de Banco de Dados*. Fortaleza: SBC, 2019, pp. 133–144. DOI: 10.5753/sbbd.2019.8814. URL: https://sol.sbc.org.br/index.php/sbbd/article/view/8814.

[145] Rodrigo Laigner, Zhexiang Zhang, Yijian Liu, Leonardo Gomes, and Yongluan Zhou. *A Benchmark for Data Management Challenges in Microservices (Extended Version)*. 2023. URL: https://rnlaigner.github.io/files/extended.pdf.

[146] Rodrigo Laigner, Zhexiang Zhang, Yijian Liu, Leonardo Freitas Gomes, and Yongluan Zhou. "Online Marketplace: A Benchmark for Data Management in Microservices". In: *Proc. ACM Manag. Data* 3.1 (Feb. 2025), pp. 1–26. DOI: 10.1145/3709653. URL: https://doi.org/10.1145/3709653.

[147] Rodrigo Laigner and Yongluan Zhou. "Benchmarking Data Management Systems for Microservices". In: *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 2024, pp. 5671–5672. DOI: 10.1109/ICDE60146.2024.00467. URL: https://doi.org/10.1109/ICDE60146.2024.00467.

[148] Rodrigo Laigner and Yongluan Zhou. *vMODB: Unifying event and data management for distributed asynchronous applications*. Manuscript under review. 2025. URL: https://arxiv.org/abs/2504.19757.

[149] Rodrigo Laigner, Yongluan Zhou, and Marcos Antonio Vaz Salles. "A Distributed Database System for Event-Based Microservices". In: *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*. DEBS '21. Virtual Event, Italy: Association for Computing Machinery, 2021, pp. 25–30. ISBN: 9781450385558. DOI: 10.1145/3465480.3466919. URL: https://doi.org/10.1145/3465480.3466919.

[150] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. "Data management in microservices: state of the practice, challenges, and research directions". In: *Proc. VLDB Endow.* 14.13 (Sept. 2021), pp. 3348–3361. ISSN: 2150-8097. DOI: 10.14778/3484224.3484232. URL: https://doi.org/10.14778/3484224.3484232.

[151] Anna Lesniak, Rodrigo Laigner, and Yongluan Zhou. "Enforcing Consistency in Microservice Architectures through Event-Based Constraints". In: *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*. DEBS '21. Virtual Event, Italy: Association for Computing Machinery, 2021, pp. 180–183. ISBN: 9781450385558. DOI: 10.1145/3465480.3467839. URL: https://doi.org/10.1145/3465480.3467839.

[152] Tianyu Li, Badrish Chandramouli, Sebastian Burckhardt, and Samuel Madden. "DARQ Matter Binds Everything: Performant and Composable Cloud Programming via Resilient Steps". In: *Proc. ACM Manag. Data* 1.2 (June 2023). DOI: 10.1145/3589262. URL: https://doi.org/10.1145/3589262.

[153] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. "The Serverless Computing Survey: A Technical Primer for Design Architecture". In: *ACM Comput. Surv.* 54.10s (Sept. 2022). ISSN: 0360-0300. DOI: 10.1145/3508360. URL: https://doi.org/10.1145/3508360.

[154] B. Liskov and L. Shrira. "Promises: linguistic support for efficient asynchronous procedure calls in distributed systems". In: *SIGPLAN*

Not. 23.7 (June 1988), pp. 260–267. ISSN: 0362-1340. DOI: 10.1145/960116.54016. URL: https://doi.org/10.1145/960116.54016.

[155] David H. Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. "Doing More with Less: Orchestrating Serverless Applications without an Orchestrator". In: *NSDI*. Apr. 2023. URL: https://www.microsoft.com/en-us/research/publication/doing-more-with-less-orchestrating-serverless-applications-without-an-orchestrator/.

[156] Yijian Liu, Rodrigo Laigner, and Yongluan Zhou. "Rethinking State Management in Actor Systems for Cloud-Native Applications". In: *Proceedings of the 2024 ACM Symposium on Cloud Computing*. SoCC '24. Redmond, WA, USA: Association for Computing Machinery, 2024, pp. 898–914. ISBN: 9798400712869. DOI: 10.1145/3698038.3698540. URL: https://doi.org/10.1145/3698038.3698540.

[157] Yijian Liu, Li Su, Vivek Shah, Yongluan Zhou, and Marcos Antonio Vaz Salles. "Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 65–78. ISBN: 9781450392495. DOI: 10.1145/3514221.3526172. URL: https://doi.org/10.1145/3514221.3526172.

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

[159] Robert Love. *Linux System Programming*. O'Reilly Media, Inc., 2007. ISBN: 9780596009588. URL: https://www.oreilly.com/library/view/linux-system-programming/0596009585/ch04.html.

[160] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. "Epoch-based commit and replication in distributed OLTP databases". In: *Proc. VLDB Endow.* 14.5 (Jan. 2021), pp. 743–756. ISSN: 2150-8097. DOI: 10.14778/3446095.3446098. URL: https://doi.org/10.14778/3446095.3446098.

[161] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. "Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '21. Seattle, WA, USA: Association for Computing Machinery, 2021, pp. 412–426. ISBN: 9781450386388. DOI: 10.1145/3472883.3487003. URL: https://doi.org/10.1145/3472883.3487003.

[162] Welder Luz, Everton Agilar, Marcos César de Oliveira, Carlos Eduardo R. de Melo, Gustavo Pinto, and Rodrigo Bonifácio. "An Experience Report on the Adoption of Microservices in Three Brazilian Government Institutions". In: *Proceedings of the XXXII Brazilian Symposium on Software Engineering*. SBES '18. Sao Carlos, Brazil: ACM, 2018, pp. 32–41. ISBN: 978-1-4503-6503-1. DOI: 10.1145/3266237.3266262.

[163] Arthur M. Del Esposte, Fabio Kon, Fabio M. Costa, and Nelson Lago. "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*. SMARTGREENS 2017. Porto, Portugal: SCITEPRESS - Science and Technology Publications, Lda, 2017, pp. 35–46. ISBN: 9789897582417. DOI: 10.5220/0006306200350046. URL: https://doi.org/10.5220/0006306200350046.

[164] Materialize. *How Ecommerce Company Drizly uses Materialize for Real-Time Notifications, Alerting, and Personalization*. URL: https://materialize.com/customer-stories/drizly.

[165] Math.NET. *Math.NET Numerics*. 2023. URL: https://numerics.mathdotnet.com/.

[166] Ricardo Mayerhofer. *restQL: Tackling microservice query complexity*. B2W, 2018. URL: https://medium.com/b2w-engineering-en/restql-tackling-microservice-query-complexity-27def5d09b40.

[167] M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giaretta, S. T. Larsen, and S. Dustdar. "Microservices: Migration of a Mission Critical System". In: *IEEE Transactions on Services Computing* (2018), pp. 1–1.

[168] Yuan Mei. *Enabling Flink's Cloud-Native Future: Introducing Disaggregated State in Flink 2.0*. 2024. URL: https://current.confluent.io/2024-sessions/enabling-flinks-cloud-native-future-introducing-disaggregated-state-in-flink-2-0.

[169] Antonio Messina, Riccardo Rizzo, Pietro Storniolo, Mario Tripiciano, and Alfonso Urso. "The Database-is-the-Service Pattern for Microservice Architectures". In: *Information Technology in Bio- and Medical Informatics*. Vol. 9832. Cham: Springer International Publishing, Sept. 2016, pp. 223–233. ISBN: 978-3-319-43948-8. DOI: 10.1007/978-3-319-43949-5_18.

[170] Microsoft. *Cluster management in Orleans*. 2023. URL: https://learn.microsoft.com/en-us/dotnet/orleans/implementation/cluster-management.

[171] Microsoft. *Dapr Distributed Application Runtime*. 2019. URL: https://github.com/dapr/dapr.

[172] Microsoft. *Grain persistence*. 2023. URL: https://learn.microsoft.com/en-us/dotnet/orleans/grains/grain-persistence/?pivots=orleans-7-0.

[173] Microsoft. *Language Integrated Query (LINQ)*. 2023. URL: https://learn.microsoft.com/en-us/dotnet/csharp/linq.

[174] Microsoft. *Why Orleans Streams?* URL: https://dotnet.github.io/orleans/Documentation/streaming/streams%5C_why.html.

[175] John Nagle. *Congestion Control in IP/TCP Internetworks*. URL: https://datatracker.ietf.org/doc/html/rfc896.

[176] Faisal Nawab and Mohammad Sadoghi. "Consensus in Data Management: From Distributed Commit to Blockchain". In: *Foundations and Trends® in Databases* 12.4 (2023), pp. 221–364. ISSN: 1931-7883. DOI: 10.1561/1900000075. URL: http://dx.doi.org/10.1561/1900000075.

[177] Davide Neri, Jacopo Soldani, Olaf Zimmermann, and Antonio Brogi. "Design principles, architectural smells and refactorings for microservices: a multivocal review". In: *SICS Software-Intensive Cyber-Physical Systems* 35.1-2 (Sept. 2019), pp. 3–15. ISSN: 2524-8529. DOI: 10.1007/s00450-019-00407-8.

[178] Osvaldo Santana Neto. *Olist Architecture From Monolith to microservices*. Olist, 2018. URL: https://www.slideshare.net/osantana/olist-architecture-v20.

[179] Sam Newman. *Building Microservices*. 1st. O'Reilly Media, Inc., 2015. ISBN: 1491950358.

[180] Sam Newman. *Pattern: Backends For Frontends*. 2015. URL: https://samnewman.io/patterns/architectural/bff/.

[181] Patrick O'Neil. "Escrow Transactions". In: *Encyclopedia of Database Systems, Second Edition*. Ed. by Ling Liu and M. Tamer Özsu. Springer, 2018. DOI: 10.1007/978-1-4614-8265-9_150. URL: https://doi.org/10.1007/978-1-4614-8265-9%5C_150.

[182] Olist, Andre Sionek, Leo Dabague, and Francisco Magioli. *Brazilian E-Commerce Public Dataset by Olist*. Olist, 2024. URL: https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce.

[183] Matheus Oliveira. *Gerenciamento de dados com micro-serviços e a experiência do iFood*. iFood, 2017. URL: https://www.infoq.com/br/presentations/gerenciamento-de-dados-com-micro-servicos-e-a-experiencia-do-ifood.

[184] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 305–319. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[185] OpenJDK. *More New I/O APIs for the Java Platform*. URL: https://openjdk.org/projects/nio/.

[186] Oracle. *Introduction to the Java Persistence API*. 2013. URL: https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html.

[187] Oracle. *Java JDBC API*. 2014. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/.

[188] Oracle. *Java Native Interface Specification—Contents*. 2024. URL: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html.

[189] Oracle. *Understanding the Two-Tier Architecture*. https://docs.oracle.com/cd/B25221_01/web.1013/b13593/undtldev011.htm. 2006.

[190] Orleans. *Best Practices*. URL: https://dotnet.github.io/orleans/docs/resources/best%5C_practices.html.

[191] Orleans. *Orleans Transactions*. https://dotnet.github.io/orleans/docs/grains/transactions.html. July 2021.

[192] Microsoft Orleans. *Grain placement*. Microsoft, 2024. URL: https://learn.microsoft.com/en-us/dotnet/orleans/grains/grain-placement.

[193] Microsoft Orleans. *Orleans streams implementation details*. Microsoft, 2024. URL: https://learn.microsoft.com/en-us/dotnet/orleans/implementation/streams-implementation/.

[194] Microsoft Orleans. *Request scheduling*. 2023. URL: https://learn.microsoft.com/en-us/dotnet/orleans/grains/request-scheduling.

[195] Microsoft Orleans. *Streaming with Orleans*. Microsoft, 2024. URL: https://learn.microsoft.com/en-us/dotnet/orleans/streaming/?pivots=orleans-7-0.

[196] Michiel Overeem, Marten Spoor, Slinger Jansen, and Sjaak Brinkkemper. "An empirical characterization of event sourced systems and their schema evolution — Lessons from industry". In: *Journal of Systems and Software* 178 (2021), p. 110970. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2021.110970. URL: https://www.sciencedirect.com/science/article/pii/S0164121221000674.

[197] Xi Pang and Jianguo Wang. "Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases". In: *Proceedings of ACM Conference on Management of Data (SIGMOD)*. 2024.

[198] Christos H Papadimitriou. "The serializability of concurrent database updates". In: *Journal of the ACM (JACM)* 26.4 (1979), pp. 631–653.

[199] Andrew Pavlo. "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*. 2017, pp. 3–3.

[200] Ilaria Pigazzini, Francesca Arcelli Fontana, Valentina Lenarduzzi, and Davide Taibi. "Towards Microservice Smells Detection". In: *Proceedings of the 3rd International Conference on Technical Debt*. TechDebt '20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 92–97. ISBN: 9781450379601. DOI: 10.1145/3387906.3388625. URL: https://doi.org/10.1145/3387906.3388625.

[201] Esmond Pitt and Kathy McNiff. *Java.rmi: The Remote Method Invocation Guide*. USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0201700433.

[202] Google Cloud Platform. *Online Boutique*. URL: https://github.com/GoogleCloudPlatform/microservices-demo.

[203] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. "Transactional Consistency and Automatic Management in an Application Data Cache". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 279–292.

[204] Dan Pritchett. "Base: An Acid Alternative". In: *File Systems and Storage*. Vol. 6. ACM Queue, 2008.

[205] Kyriakos Psarakis, George Christodoulou, Marios Fragkoulis, and Asterios Katsifodimos. "Transactional Cloud Applications Go with the (Data)Flow". In: *15th Annual Conference on Innovative Data Systems Research (CIDR'25). January 19-22, 2025, Amsterdam, The Netherlands.* 2025.

[206] Kyriakos Psarakis, George Siachamis, George Christodoulou, Marios Fragkoulis, and Asterios Katsifodimos. *Styx: Transactional Stateful Functions on Streaming Dataflows*. 2024. arXiv: 2312.06893 [cs.DC]. URL: https://arxiv.org/abs/2312.06893.

[207] Kyriakos Psarakis, Wouter Zorgdrager, Marios Fragkoulis, Guido Salvaneschi, and Asterios Katsifodimos. "Stateful entities: object-oriented cloud applications as distributed dataflows". In: *Proceedings of the 27th International Conference on Extending Database Technology (EDBT)* (2023), pp. 15–21.

[208] RabbitMQ. *RabbitMQ*. (Accessed on 16/12/2024). URL: https://www.rabbitmq.com/.

[209] M. Ramachandran and Z. Mahmood. *Software Engineering in the Era of Cloud Computing*. Computer Communications and Networks. Springer International Publishing, 2020. ISBN: 9783030336240. URL: https://books.google.dk/books?id=v5PHDwAAQBAJ.

[210] Redis. 2020. URL: https://redis.io/blog/diving-into-redis-6.

[211] Redis. *Redis Streams*. URL: https://redis.io/docs/latest/develop/data-types/streams/.

[212] RedPanda. *RedPanda*. (Accessed on 16/12/2024). URL: https://www.redpanda.com/.

[213] Mark Richards. *Software Architecture Patterns*. 1st. O'Reilly, 2015.

[214] Chris Richardson. *API gateway pattern*. URL: https://microservices.io/patterns/apigateway.html.

[215] Chris Richardson. *ftgo-application*. URL: https://github.com/microservices-patterns/ftgo-application.

[216] D. Richter, M. Konrad, K. Utecht, and A. Polze. "Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice". In: *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 2017, pp. 130–137.

[217] RisingWave. *RisingWave vs Apache Flink*. URL: https://risingwave.com/risingwave-vs-apache-flink.

[218] Nitin Sarma. *Event Driven Architectures @ Netflix Content Finance Engineering*. Netflix, 2020. URL: https://www.linkedin.com/pulse/event-driven-architectures-netflix-content-finance-engineering-s/.

[219] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. "Closed versus open system models and their impact on performance and scheduling". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2006.

[220] Sentilo. *Sentilo Platform*. URL: https://github.com/sentilo/sentilo.

[221] Vishwanath Seshagiri, Darby Huye, Lan Liu, Avani Wildani, and Raja Sambasivan. "[SoK] Identifying Mismatches Between Microservice Testbeds and Industrial Perceptions of Microservices". In: *Journal of Systems Research* 2 (Mar. 2022). DOI: 10.5070/SR32157839.

[222] Vivek Shah. "Exploration of a Vision for Actor Database Systems". PhD thesis. University of Copenhagen, Denmark, 2017, p. 121.

[223] Vivek Shah and Marcos Antonio Vaz Salles. "Reactors: A Case for Predictable, Virtualized Actor Database Systems". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 259–274. ISBN: 9781450347037. DOI: 10.1145/3183713.3183752. URL: https://doi.org/10.1145/3183713.3183752.

[224] Natan Silnitsky. *Event Driven Architecture — 5 Pitfalls to Avoid*. Wix, 2022. URL: https://medium.com/wix-engineering/event-driven-architecture-5-pitfalls-to-avoid-b3ebf885bdb1.

[225] Natan Silnitsky. *Troubleshooting Kafka for 2000 Microservices at Wix*. Wix, 2022. URL: https://medium.com/wix-engineering/troubleshooting-kafka-for-2000-microservices-at-wix-986ee382fd1e.

[226] Aakriti Singla and Simon Wu. *Revolutionizing Money Movements at Scale with Strong Data Consistency*. Uber Technologies Inc., 2020. URL: https://eng.uber.com/money-scale-strong-data.

[227] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. "The pains and gains of microservices: A Systematic grey literature review". In: *Journal of Systems and Software* 146 (2018), pp. 215–232.

[228] CAE Specification. *Distributed Transaction Processing: the XA Specification*. X/Open, 1991.

[229] Jonas Spenger, Chengyang Huang, Philipp Haller, and Paris Carbone. "Portals: A Showcase of Multi-Dataflow Stateful Serverless". In: *Proc. VLDB Endow.* 16.12 (Aug. 2023), pp. 4054–4057. ISSN: 2150-8097. DOI: 10.14778/3611540.3611619. URL: https://doi.org/10.14778/3611540.3611619.

[230] Spring. *Java Spring*. (Accessed on 16/12/2024). URL: https://spring.io/.

[231] Spring. *Message Listener Containers*. 2024. URL: https://docs.spring.io/spring-kafka/reference/kafka/receiving-messages/message-listener-container.html%5C#committing-offsets.

[232] Spring. *spring-petclinic-microservices*. URL: https://github.com/spring-petclinic/spring-petclinic-microservices.

[233] Spring.io. *Spring Framework*. URL: http://spring.io.

[234] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. "Cloudburst: stateful functions-as-a-service". In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 2438–2452. ISSN: 2150-8097. DOI: 10.14778/3407790.3407836. URL: https://doi.org/10.14778/3407790.3407836.

[235] StackOverflow. *2023 Developer Survey*. 2023. URL: https://survey.stackoverflow.co/2023/.

[236] Apache Flink Statefun. *Co-located Functions*. 2023. URL: https://nightlies.apache.org/flink/flink-statefun-docs-master/docs/concepts/distributed_architecture.

[237] Apache Flink Statefun. *Distributed Architecture*. 2023. URL: https://nightlies.apache.org/flink/flink-statefun-docs-master/docs/concepts/distributed_architecture.

[238] Apache Flink Statefun. *Shopping Cart Example with Docker Compose*. 2023. URL: https://github.com/apache/flink-statefun-playground/tree/main/java/shopping-cart.

[239] Apache Flink Statefun. *Stateful Functions: A Platform-Independent Stateful Serverless Stack*. 2020. URL: https://nightlies.apache.org/flink/flink-statefun-docs-master/.

[240] Statista. *eCommerce - Worldwide*. 2024. URL: https://www.statista.com/outlook/emo/ecommerce/worldwide.

[241] Data Stax. *Astra DB Multi-cloud DBaaS built on Apache Cassandra*. URL: https://www.datastax.com/products/datastax-astra.

[242] Jan Stenberg. *Experiences Moving from Microservices to Workflows at Jet.com*. Jet.com, 2019. URL: https://www.infoq.com/news/2019/02/migrate-microservices-workflows.

[243] Mirko Stocker and Stefan Kapferer. *LakesideMutual*. URL: https://github.com/Microservice-API-Patterns/LakesideMutual.

[244] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. "P-Store: An Elastic Database System with Predictive Provisioning". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 205–219. ISBN: 9781450347037. DOI: 10.1145/3183713.3190650. URL: https://doi.org/10.1145/3183713.3190650.

[245] Tzu-Li (Gordon) Tai. *Stateful Functions Internals: Behind the scenes of Stateful Serverless*. 2020. URL: https://flink.apache.org/news/2020/10/13/stateful-serverless-internals.html.

[246] D. Taibi and V. Lenarduzzi. "On the Definition of Microservice Bad Smells". In: *IEEE Software* 35.3 (2018), pp. 56–62.

[247] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. "Microservices Anti-patterns: A Taxonomy". In: *Microservices: Science and Engineering*. Cham: Springer International Publishing, 2020, pp. 111–128. ISBN: 978-3-030-31646-4. DOI: 10.1007/978-3-030-31646-4_5. URL: https://doi.org/10.1007/978-3-030-31646-4%5C_5.

[248] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. 2nd. CreateSpace Independent Publishing Platform, 2016.

[249] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. "Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 4–18. ISBN: 9781450392495. DOI: 10.1145/3514221.3526120. URL: https://doi.org/10.1145/3514221.3526120.

[250] Alexander Thomson and Daniel J Abadi. "The case for determinism in database systems". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 70–80.

[251] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. "Calvin: fast distributed transactions for partitioned database systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, pp. 1–12.

[252] Rafik Tighilt, Manel Abdellatif, Naouel Moha, Hafedh Mili, Ghizlane El Boussaidi, Jean Privat, and Yann-Gaël Guéhéneuc. "On the Study of Microservices Antipatterns: a Catalog Proposal". In: *Proceedings of the European Conference on Pattern Languages of Programs 2020*. EuroPLoP '20. Virtual Event, Germany: Association for Computing Machinery, 2020. ISBN: 9781450377690. DOI: 10.1145/3424771.3424812. URL: https://doi.org/10.1145/3424771.3424812.

[253] Lillian Tsai, Hannah Gross, Eddie Kohler, Frans Kaashoek, and Malte Schwarzkopf. "Edna: Disguising and Revealing User Data in Web Applications". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. Koblenz, Germany: Association for Computing Machinery, 2023, pp. 434–450. ISBN: 9798400702297. DOI: 10.1145/3600006.3613146. URL: https://doi.org/10.1145/3600006.3613146.

[254] UCloud. *Products - UCloud*. URL: https://docs.cloud.sdu.dk/guide/resources-products.html.

[255] UCloud. *UCloud User Guide*. URL: https://docs.cloud.sdu.dk.

[256] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. "Benchmarking, analysis, and optimization of serverless function snapshots". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 559–572. ISBN: 9781450383172. DOI: 10.1145/3445814.3446714. URL: https://doi.org/10.1145/3445814.3446714.

[257] Panos Vassiliadis. "A survey of extract–transform–load technology". In: *International Journal of Data Warehousing and Mining (IJDWM)* 5.3 (2009), pp. 1–27.

[258] Miguel Veloso. *BFF implementation*. 2019. URL: https://github.com/dotnet-architecture/eShopOnContainers/wiki/BFF-implementation.

[259] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 1041–1052. ISBN: 9781450341974. DOI: 10.1145/3035918.3056101. URL: https://doi.org/10.1145/3035918.3056101.

[260] Nicolas Viennot, Mathias Lécuyer, Jonathan Bell, Roxana Geambasu, and Jason Nieh. "Synapse: A Microservices Architecture for Heterogeneous-Database Web Applications". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450332385. DOI: 10.1145/2741948.2741975. URL: https://doi.org/10.1145/2741948.2741975.

[261] Markos Viggiato, Ricardo Terra, Henrique Rocha, Marco Tulio Valente, and Eduardo Figueiredo. "Microservices in Practice: A Survey Study". In: *arXiv e-prints*, arXiv:1808.04836 (Aug. 2018). arXiv: 1808.04836 [cs.SE].

[262] Steve Vinoski. "CORBA: Integrating diverse applications within distributed heterogeneous environments". In: *IEEE Communications magazine* 35.2 (1997), pp. 46–55.

[263] VTEX. *Commerce Microservices*. VTEX. URL: https://vtex.com/us-en/commerce-microservices/.

[264] Jianguo Wang and Qizhen Zhang. "Disaggregated Database Systems". In: *Companion of the 2023 International Conference on Management of Data*. SIGMOD '23. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 37–44. ISBN: 9781450395076. DOI: 10.1145/3555041.3589403. URL: https://doi.org/10.1145/3555041.3589403.

[265] Yiwen Wang, Júlio César dos Reis, Kasper Myrtue Borggren, Marcos Antonio Vaz Salles, Claudia Bauzer Medeiros, and Yongluan Zhou. "Modeling and Building IoT Data Platforms with Actor-Oriented Databases". In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon,*

Portugal, March 26-29, 2019. Ed. by Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi. OpenProceedings.org, 2019, pp. 512–523. DOI: 10.5441/002/edbt.2019.47. URL: https://doi.org/10.5441/002/edbt.2019.47.

[266] Weaveworks. *Sock Shop : A Microservice Demo Application*. URL: https://github.com/microservices-demo/microservices-demo.

[267] Edwin van Wijk. *pitstop*. URL: https://github.com/EdwinVW/pitstop.

[268] Marianne Winslett and Vanessa Braganholo. "Peter Bailis Speaks Out on Building Tools Users Want to Use". In: *SIGMOD Rec.* 47.3 (Feb. 2019), pp. 29–31. ISSN: 0163-5808. DOI: 10.1145/3316416.3316423. URL: https://doi.org/10.1145/3316416.3316423.

[269] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. "An empirical evaluation of in-memory multi-version concurrency control". In: *Proc. VLDB Endow.* 10.7 (Mar. 2017), pp. 781–792. ISSN: 2150-8097. DOI: 10.14778/3067421.3067427. URL: https://doi.org/10.14778/3067421.3067427.

[270] Fan Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. "Hilda: A High-Level Language for Data-DrivenWeb Applications". In: *22nd International Conference on Data Engineering (ICDE'06)*. 2006, pp. 32–32. DOI: 10.1109/ICDE.2006.75.

[271] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. "PowerStation: Automatically Detecting and Fixing Inefficiencies of Database-Backed Web Applications in IDE". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 884–887. ISBN: 9781450355735. DOI: 10.1145/3236024.3264589. URL: https://doi.org/10.1145/3236024.3264589.

[272] Yang Yang, Zhifeng Chen, Qichao Chu, Haitao Zhang, and George Teo. *Enabling Seamless Kafka Async Queuing with Consumer Proxy*. Uber Technologies Inc., 2021. URL: https://www.uber.com/en-SE/blog/kafka-async-queuing-with-consumer-proxy/.

[273] Stephen Yau and Ho An. "Software engineering meets services and cloud computing". In: *Computer* 44.10 (2011), pp. 47–53.

[274] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. "DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 1–14.

[275] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized streams: fault-tolerant streaming computation at scale". In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. 2013, pp. 423–438.

[276] Chao Zhang, Guoliang Li, and Tao Lv. "HyBench: A New Benchmark for HTAP Databases". In: *Proc. VLDB Endow.* 17.5 (May 2024), pp. 939–951. ISSN: 2150-8097. DOI: 10.14778/3641204.3641206. URL: https://doi.org/10.14778/3641204.3641206.

[277] Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. "UniBench: A Benchmark for Multi-model Database Management Systems". In: *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence*. Ed. by Raghunath Nambiar and Meikel Poess. Cham: Springer International Publishing, 2019, pp. 7–23. ISBN: 978-3-030-11404-6.

[278] H. Zhang, S. Li, Z. Jia, C. Zhong, and C. Zhang. "Microservice Architecture in Reality: An Industrial Inquiry". In: *2019 IEEE International Conference on Software Architecture (ICSA)*. 2019, pp. 51–60.

[279] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. "Fault-tolerant and transactional stateful serverless workflows". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1187–1204. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/zhang-haoran.

[280] Haoran Zhang, Konstantinos Kallas, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, and Vincent Liu. "MuCache: A General Framework for Caching in Microservice Graphs". In: *21st USENIX Symposium*

on Networked Systems Design and Implementation (NSDI 24). Santa Clara, CA: USENIX Association, Apr. 2024, pp. 221–238. ISBN: 978-1-939133-39-7. URL: https://www.usenix.org/conference/nsdi24/presentation/zhang-haoran.

[281] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. "The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 195–211. ISBN: 9781450387095. DOI: 10.1145/3477132.3483569. URL: https://doi.org/10.1145/3477132.3483569.

[282] Jianqiu Zhang, Kaisong Huang, Tianzheng Wang, and King Lv. "Efficiently Making Cross-Engine Transactions Consistent". In: *SIGMOD Rec.* 52.1 (June 2023), pp. 27–34. ISSN: 0163-5808. DOI: 10.1145/3604437.3604444. URL: https://doi.org/10.1145/3604437.3604444.

[283] Shuhao Zhang, Juan Soto, and Volker Markl. "A survey on transactional stream processing". In: *The VLDB Journal* 33.2 (2024), pp. 451–479.

[284] Eric Zhao. *vertx-blueprint-microservice*. URL: https://github.com/sczyh30/vertx-blueprint-microservice.

[285] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study". In: *IEEE Trans. Softw. Eng.* 47.2 (Feb. 2021), pp. 243–260. ISSN: 0098-5589. DOI: 10.1109/TSE.2018.2887384. URL: https://doi.org/10.1109/TSE.2018.2887384.

[286] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. "Benchmarking microservice systems for software engineering research". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman. ACM,

2018, pp. 323–324. DOI: 10.1145/3183440.3194991. URL: https://doi.org/10.1145/3183440.3194991.

[287] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. "Dissecting Overheads of Service Mesh Sidecars". In: *Proceedings of the 2023 ACM Symposium on Cloud Computing*. SoCC '23. Santa Cruz, CA, USA: Association for Computing Machinery, 2023, pp. 142–157. ISBN: 9798400703874. DOI: 10.1145/3620678.3624652. URL: https://doi.org/10.1145/3620678.3624652.

[288] Olaf Zimmermann. "Microservices Tenets". In: *Comput. Sci.* 32.3-4 (2017), pp. 301–310.