Transactional Cloud Applications: Status Quo, Challenges, and Opportunities

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

George Christodoulou
Delft University of Technology
Delft, Netherlands
g.c.christodoulou@tudelft.nl

Kyriakos Psarakis
Delft University of Technology
Delft, Netherlands
k.psarakis@tudelft.nl

Asterios Katsifodimos
Delft University of Technology
Delft, Netherlands
a.katsifodimos@tudelft.nl

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

Abstract

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 tutorial aims to highlight recent trends in the area and discusses open research challenges for the data management community.

CCS Concepts

• **Information systems** → **Data management systems**; • **Computer systems organization** → **Distributed architectures**.

Keywords

transaction processing, data management, cloud computing

ACM Reference Format:
Rodrigo Laigner, George Christodoulou, Kyriakos Psarakis, Asterios Katsifodimos, and Yongluan Zhou. 2025. Transactional Cloud Applications: Status Quo, Challenges, and Opportunities. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3722212.3725635

SIGMOD-Companion '25, June 22–27, 2025, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725635

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) [13]. 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 [16], Akka Serverless) and the serverless computing paradigm, such as AWS Lambda [10], 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 [47].

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. With this tutorial, 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.

2 Tutorial Overview

In this tutorial, we propose a taxonomy (section 3) 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.

Tutorial Outline (3 hours)

- **Context and motivation (45 minutes)**
  - Introduction and context
  - From monoliths to cloud-native applications
- **Building blocks (45 minutes)**
  - Programming models
  - State management
  - Messaging
- **Requirements (45 minutes)**
  - Fault-tolerance
  - Application lifecycle
  - Scalability
- **Future directions (45 minutes)**
  - Open problems
  - Research opportunities

Target Audience. The target audience of this tutorial includes PhD students, researchers, and practitioners of different roles (e.g., software and data architects and engineers) who intend to obtain a clear overview of the state-of-the-art cloud application development landscape and its implications for data management. The tutorial is self-contained and provides the background on scalable cloud applications; no prior knowledge of systems for cloud applications is required. However, familiarity with application architectures helps one understand the materials.

3 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 1: *i*) programming models (§ 3.1), with a focus on their parallelization primitives and how practitioners are building such applications; *ii*) messaging (§ 3.2) with focus on different ways of exchanging messages and performing remote procedure calls and finally; *iii*) state management (§ 3.3), with a focus on transactions and state consistency across services and scalability. Their interplay leads to trade-offs concerning programmability, consistency, and performance, as recent findings suggest [19, 39].

3.1 Programming Cloud Applications

Programming models for distributed systems has been a long-standing line of research [9, 11, 12, 65, 67]. 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 [3], Python Flask [4]) and emerging programming models, namely Actors (e.g., Akka [1], Orleans [16]) and Stateful Functions (e.g., Flink Statefun [59], Azure Durable Functions [15]), all differing significantly in system model, abstractions, and guarantees offered to developers.

Therefore, we kick off this tutorial by 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 tutorial 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 [50], called microservices. Microservice frameworks, such as Spring Boot (Java) [3], Flask (Python) [4], and Dapr (C#/.NET) [2], 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 [8]. 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

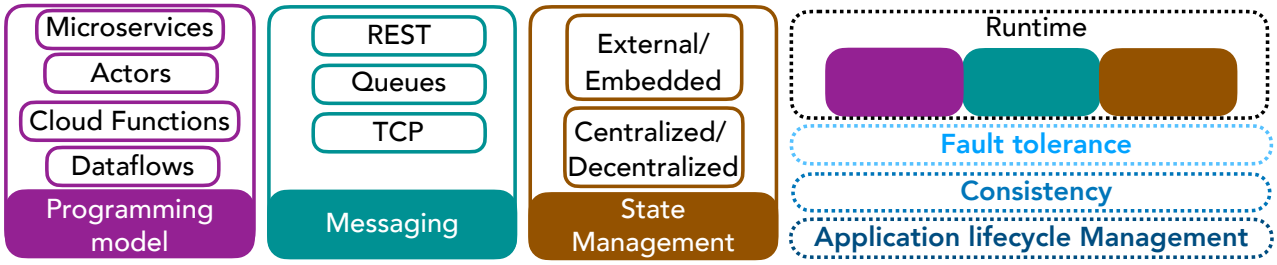

Figure 1: Building blocks and requirements for transactional cloud applications.

asynchronous message-passing. Concurrency in actor systems is achieved by pipelining and dynamic creation of actors [8]. Traditional actor systems allow programmers to develop systems using low-level primitives by using actor IDs and prescribing their physical locations.

Virtual actors [16] 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 [16] and Dapr [2].

Cloud Functions. With the emergence of serverless computing [41], a new cloud paradigm called Function-as-a-Service (FaaS) [15, 61] 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 [58, 69]. More recently, though, there has been increasing interest in extending the FaaS paradigm to applications that access state intensively, called Stateful-FaaS (SFaaS) [21, 35, 52, 58, 69]. 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 [17]. 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 [22, 51, 59, 71].

3.2 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 [48] or CORBA's OMG [63]. 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 [34] 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` [30]) is traditionally leveraged to prevent the execution of non-idempotent operations for incoming 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 [24], adding to the complexity of developing cloud applications.

Message Queues. Message queues (e.g., Apache Kafka [37], RabbitMQ [5], RedPanda [6], 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 [60]. 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 [39].

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.

3.3 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 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 [39]: *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 (§ 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 [42]. 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 [45]. 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 [16]. 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 [58]. 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 [41]. 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 [17]. Typically, operators are scheduled for execution in separate nodes and rely on embedded LSM-based key-value stores like RocksDB [23] 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 [44, 55]. In this case, cloud object storage systems like S3 are used not only for checkpointing states [17] but also to store operators' states. It is worth noting that, unlike service-based architectures, state management in dataflow systems is transparent to developers.

3.4 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 [31]. 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.

4 Requirements

In this section, we discuss the cross-cutting requirements of transactional cloud applications, namely: *i*) fault-tolerance (§ 4.1); *ii*) consistency (§ 4.2) and *iii*) application lifecycle management (§ 4.3).

4.1 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 [16, 42], 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 [16]. However, weak message delivery semantics and lack of transactional guarantees can leave actor states inconsistent after failures (§ 4.2).

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 [18]. 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.

4.2 Consistency

The consistency models in distributed systems reason about reads and writes on shared state and their real-time order guarantees across processes [60]. 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 [50], characterized by eventually consistent application partitions through queuing operations [33]. Practitioners also refer to this eventual consistency model through sagas [28] or patterns like orchestration and workflows [7].

Microservices often avoid distributed commit protocols to decouple components [39]. 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 [36]. 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.

To enable transactional serializability in Orleans, users must utilize the Transactions API [46]. Apart necessitating porting the actor attributes to opaque objects [42], it has been shown to introduce a significant performance penalty according to recent experimental evaluations [38, 43], demotivating broader adoption.

Cloud Functions. Cloudburst enriches functions with causally consistent shared state accesses through a key-value abstraction [58]. Durable functions [14], 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 isolation across functions.

Another category of Cloud Function systems goes beyond by providing transactional serializability on computations cutting across functions [35, 69]. However, recent work [52] 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 [17, 68]. 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 cannot ensure transactional isolation.

4.3 Application Lifecycle Management

Resource Management. The programming abstractions offered to developers (§ 3.1) also play a key role in application lifecycle management. 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 (§ 3.3). These challenges motivated the development of distributed frameworks that transparently manage the lifecycle of applications. Serverless computing and FaaS offer transparent resource provisioning, function scheduling, failure handling, and elasticity to application maintainers. However, challenges associated with cold starts, execution performance, and costs undermine a wider adoption of the FaaS paradigm in application architectures [41].

Application Evolution. The evolution of applications is a key concern in the software engineering lifecycle, and it is no different in cloud applications [54, 66]. 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. In this tutorial, we cover the application evolution space for service-oriented architectures, actors, and dataflow systems.

5 Open Problems & Research Opportunities

In this section, we describe a set of open problems in programming models (§ 5.1), state and messaging (§ 5.2) and benchmarks (§ 5.3).

5.1 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* [7, 25, 59], *workflows* [2, 7], *durable* [7], *stateful* [7, 25, 52, 59], *reliable* [25], and *virtual* [2, 16], which are not consistent across systems since they express varied guarantees for applications.

Apart from the dataflow and actor models [8], and more recently Durable Functions [15], 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 [53], HydroLogic [20], and event-based constraints [40].

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.

5.2 State & Messaging

Data Model. Programming models used in the cloud often provide opaque state management abstractions [16, 32, 42, 53]. 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 [20] that are popular in practice [38], not jeopardizing state encapsulation.

Disaggregation. In the same line of industry-strength cloud-native database [62] and stream processing systems [29, 44], disaggregated storage [64] 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 [41], 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 [60], 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 [28] and OpenXA [57] allow for coordinating consistency guarantees across microservices. More recent work [26] introduces causal consistency for microservice architectures. Cross-engine transactions [70] 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.

5.3 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 [27] have been used to evaluate distributed cloud application frameworks [35, 69] alongside TPC-C [52]. In addition, despite recent efforts to benchmark cloud applications [27, 49, 72], 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 [56].

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 [38] aims to fill these gaps, but challenges related to dynamic workloads, observability, and recovery remain open.

6 Biographies

Rodrigo is a PhD Fellow at the University of Copenhagen. His research lies on devising effective programming abstractions and efficient systems for emerging data-intensive applications. During his doctoral studies, he published relevant articles about distributed data-intensive applications.

George is a Postdoctoral Researcher at TU Delft. His research centers around indexing, as well as scalable and efficient data management with a particular emphasis on stream processing, and distributed systems.

Kyriakos is a PhD candidate at TU Delft building systems for scalable cloud applications. This includes Styx, a deterministic transactional dataflow system that offers a Stateful-FaaS API for creating scalable cloud applications.

Asterios is an Asst. Professor at TU Delft, working on scalable data management, focusing on cloud application runtimes, stream processing, and data integration. Asterios is one of the receivers of the ACM SIGMOD Systems award in 2023.

Yongluan is a Professor at the University of Copenhagen. His research interests span database and distributed systems, with his recent focus on scalable event-driven systems. He has authored over 80 peer-reviewed research articles in international journals and conference proceedings.

Acknowledgments

This work was partially supported by Independent Research Fund Denmark under Grant 9041-00368B, as well as the Vidi research program project number 19708, financed by the Dutch Research Council (NWO).

References

[1] [n. d.]. *Akka.* https://akka.io/ (Accessed on 16/12/2024).

[2] [n. d.]. *Dapr - Distributed Application Runtime.* https://dapr.io/ (Accessed on 27/11/2024).

[3] [n. d.]. *Java Spring.* https://spring.io/ (Accessed on 16/12/2024).

[4] [n. d.]. *Python Flask.* https://flask.palletsprojects.com/en/stable/ (Accessed on 16/12/2024).

[5] [n. d.]. *RabbitMQ.* https://www.rabbitmq.com/ (Accessed on 16/12/2024).

[6] [n. d.]. *RedPanda.* https://www.redpanda.com/ (Accessed on 16/12/2024).

[7] David Liu , Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. 2023. Doing More with Less: Orchestrating Serverless Applications without an Orchestrator. In *NSDI*. https://www.microsoft.com/en-us/research/publication/doing-more-with-less-orchestrating-serverless-applications-without-an-orchestrator/

[8] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press. https://doi.org/10.7551/mitpress/1086.001.0001

[9] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. 2010. Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 223–236. https://doi.org/10.1145/1755913.1755937

[10] AWS. [n. d.]. *Lambda.* https://aws.amazon.com/lambda/ (Accessed on 27/11/2024).

[11] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. 1992. Orca: A language for parallel programming of distributed systems. *IEEE transactions on software engineering* 18, March (1992), 190–205.

[12] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. 1989. Programming languages for distributed computing systems. *ACM Comput. Surv.* 21, 3 (Sept. 1989), 261–322. https://doi.org/10.1145/72551.72552

[13] Philip A. Bernstein. 2019. Resurrecting Middle-Tier Distributed Transactions. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 42, 2 (June 2019), 3–6.

[14] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: efficient execution of serverless workflows. *Proc. VLDB Endow.* 15, 8 (April 2022), 1591–1604. https://doi.org/10.14778/3529337.3529344

[15] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 133 (Oct. 2021), 27 pages. https://doi.org/10.1145/3485510

[16] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 1–14. https://doi.org/10.1145/2038916.2038932

[17] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015), 28–38.

[18] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (1985), 63–75.

[19] Chaoyi Cheng, Mingzhe Han, Nuo Xu, Spyros Blanas, Michael D Bond, and Yang Wang. 2023. 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.*

[20] Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Mae Milano. 2021. 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. http://cidrdb.org/cidr2021/papers/cidr2021_paper16.pdf

[21] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Distributed transactions on serverless stateful functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems.* 31–42.

[22] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. 2022. Transactions across serverless functions leveraging stateful dataflows. *Information Systems* 108 (2022), 102015. https://doi.org/10.1016/j.is.2022.102015

[23] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (Oct. 2021), 32 pages. https://doi.org/10.1145/3483840

[24] João Esteves, Rosa Costa, Yongluan Zhou, and Ana Almeida. 2023. 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* (Neuchatel, Switzerland) *(DEBS '23)*. Association for Computing Machinery, New York, NY, USA, 91–102. https://doi.org/10.1145/3583678.3596898

[25] Azure Service Fabric. [n. d.]. *Service Fabric programming model overview.* https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-choose-framework

[26] João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. 2023. Antipode: Enforcing Cross-Service Causal Consistency in Distributed Applications. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 298–313. https://doi.org/10.1145/3600006.3613176

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

[28] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '87)*. Association for Computing Machinery, New York, NY, USA, 249–259. https://doi.org/10.1145/38713.38742

[29] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B Kahveci, Ali Gürbüz Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, Ufuk Yılmaz, et al. 2021. Hazelcast Jet: Low-latency stream processing at the 99.99 th percentile. *arXiv preprint arXiv:2103.10169* (2021).

[30] Network Working Group. 2020. *The Idempotency HTTP Header Field.* https://datatracker.ietf.org/doc/html/draft-idempotency-header-00

[31] Christopher Gustafson. 2022. *Improving Availability of Stateful Serverless Functions in Apache Flink.* Master thesis. KTH Royal Institute of Technology.

[32] Pat Helland. 2007. Life beyond Distributed Transactions: an Apostate's Opinion. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings.* www.cidrdb.org, 132–141. http://cidrdb.org/cidr2007/papers/cidr07p15.pdf

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

[34] Kasun Indrasiri and Danesh Kuruppu. 2020. *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes.* O'Reilly Media.

[35] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 691–707. https://doi.org/10.1145/3477132.3483541

[36] Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Matei Zaharia, and Xiangyao Yu. 2023. Epoxy: ACID Transactions across Diverse Data Stores. *Proc. VLDB Endow.* 16, 11 (July 2023), 2742–2754. https://doi.org/10.14778/3611479.3611484

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

[38] Rodrigo Laigner, Zhexiang Zhang, Yijian Liu, Leonardo Freitas Gomes, and Yongluan Zhou. 2025. Online Marketplace: A Benchmark for Data Management in Microservices. *Proc. ACM Manag. Data* 3, 1, Article 3 (Feb. 2025), 26 pages. https://doi.org/10.1145/3709653

[39] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *Proc. VLDB Endow.* 14, 13 (sep 2021), 3348–3361. https://doi.org/10.14778/3484224.3484232

[40] Anna Lesniak, Rodrigo Laigner, and Yongluan Zhou. 2021. Enforcing consistency in microservice architectures through event-based constraints. In *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems* (Virtual Event, Italy) *(DEBS '21)*. Association for Computing Machinery, New York, NY, USA, 180–183. https://doi.org/10.1145/3465480.3467839

[41] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. 2022. The Serverless Computing Survey: A Technical Primer for Design Architecture. *ACM Comput. Surv.* 54, 10s, Article 220 (Sept. 2022), 34 pages. https://doi.org/10.1145/3508360

[42] Yijian Liu, Rodrigo Laigner, and Yongluan Zhou. 2024. Rethinking State Management in Actor Systems for Cloud-Native Applications. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 898–914. https://doi.org/10.1145/3698038.3698540

[43] Yijian Liu, Li Su, Vivek Shah, Yongluan Zhou, and Marcos Antonio Vaz Salles. 2022. Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/3514221.3526172

[44] Yuan Mei. 2024. *Enabling Flink's Cloud-Native Future: Introducing Disaggregated State in Flink 2.0.* https://current.confluent.io/2024-sessions/enabling-flinks-cloud-native-future-introducing-disaggregated-state-in-flink-2-0

[45] Orleans. [n. d.]. *Best Practices.* Retrieved October, 15 2024 from https://dotnet.github.io/orleans/docs/resources/best_practices.html

[46] Orleans. 2021. Orleans Transactions. https://dotnet.github.io/orleans/docs/grains/transactions.html.

[47] Christos H Papadimitriou. 1979. The serializability of concurrent database updates. *Journal of the ACM (JACM)* 26, 4 (1979), 631–653.

[48] Esmond Pitt and Kathy McNiff. 2001. *Java.rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., USA.

[49] Google Cloud Platform. [n. d.]. Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo

[50] Dan Pritchett. 2008. Base: An Acid Alternative. In *File Systems and Storage*, Vol. 6. ACM Queue. Issue 3.

[51] Kyriakos Psarakis, George Christodoulou, Marios Fragkoulis, and Asterios Katsifodimos. 2025. 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*.

[52] Kyriakos Psarakis, George Christodoulou, George Siachamis, Marios Fragkoulis, and Asterios Katsifodimos. 2025. Styx: Transactional Stateful Functions on Streaming Dataflows. *Proc. ACM Manag. Data*, Article 226 (2025). https://doi.org/10.1145/3725363

[53] Kyriakos Psarakis, Wouter Zorgdrager, Marios Fragkoulis, Guido Salvaneschi, and Asterios Katsifodimos. 2023. Stateful entities: object-oriented cloud applications as distributed dataflows. *Proceedings of the 27th International Conference on Extending Database Technology (EDBT)* (2023), 15––21.

[54] M. Ramachandran and Z. Mahmood. 2020. *Software Engineering in the Era of Cloud Computing*. Springer International Publishing. https://books.google.dk/books?id=v5PHDwAAQBAJ

[55] RisingWave. [n. d.]. *RisingWave vs Apache Flink*. https://risingwave.com/risingwave-vs-apache-flink

[56] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Closed versus open system models and their impact on performance and scheduling. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[57] CAE Specification. 1991. *Distributed Transaction Processing: the XA Specification*. X/Open.

[58] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: stateful functions-as-a-service. *Proc. VLDB Endow.* 13, 12 (July 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[59] Apache Flink Statefun. 2023. *Stateful Functions: A Platform-Independent Stateful Serverless Stack*. Retrieved November 11, 2023 from https://nightlies.apache.org/flink/flink-statefun-docs-master/

[60] Andrew S. Tanenbaum and Maarten van Steen. 2008. *Distributed Systems: Principles and Paradigms* (2nd rev. ed. ed.). Prentice Hall International.

[61] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 559–572. https://doi.org/10.1145/3445814.3446714

[62] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[63] Steve Vinoski. 1997. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications magazine* 35, 2 (1997), 46–55.

[64] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 37–44. https://doi.org/10.1145/3555041.3589403

[65] Fan Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. 2006. Hilda: A High-Level Language for Data-DrivenWeb Applications. In *22nd International Conference on Data Engineering (ICDE'06)*. 32–32. https://doi.org/10.1109/ICDE.2006.75

[66] Stephen Yau and Ho An. 2011. Software engineering meets services and cloud computing. *Computer* 44, 10 (2011), 47–53.

[67] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. 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* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 1–14.

[68] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. 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*. 423–438.

[69] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1187–1204. https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

[70] Jianqiu Zhang, Kaisong Huang, Tianzheng Wang, and King Lv. 2023. Efficiently Making Cross-Engine Transactions Consistent. *SIGMOD Rec.* 52, 1 (June 2023), 27–34. https://doi.org/10.1145/3604437.3604444

[71] Shuhao Zhang, Juan Soto, and Volker Markl. 2024. A survey on transactional stream processing. *The VLDB Journal* 33, 2 (2024), 451–479.

[72] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. 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*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 323–324. https://doi.org/10.1145/3183440.3194991