Transactional Cloud Applications Go with the (Data) Flow

Psarakis, K.; Christodoulou, G.C.; Fragkoulis, M.; Katsifodimos, A

Publication date
2025

Document Version
Final published version

Published in
15th Annual Conference on Innovative Data Systems Research (CIDR '25)

Transactional Cloud Applications Go with the (Data)Flow

Kyriakos Psarakis George Christodoulou Marios Fragkoulis Asterios Katsifodimos
{k.psarakis,g.c.christodoulou,m.fragkoulis,a.katsifodimos}@tudelft.nl
Delft University of Technology

ABSTRACT

Traditional monolithic applications are migrated to the cloud, typically using a microservice-like architecture. Although this migration leads to significant benefits such as scalability and development agility, it also leaves behind the transactional guarantees that database systems have provided to monolithic applications for decades. In the cloud era, developers build transactional and fault-tolerant distributed applications by explicitly programming transaction protocols at the application level.

In this paper, we argue that the principles behind the streaming dataflow execution model and deterministic transactional protocols provide a powerful and suitable substrate for executing transactional cloud applications. To this end, we introduce Styx, a transactional application runtime based on streaming dataflows that enables an object-oriented programming model for scalable, fault-tolerant cloud applications with serializable guarantees.

1 INTRODUCTION

During the last decades, enterprises have migrated applications such as order management systems, banking systems, game-backend services, and supply-chain management to the cloud. The transition from monolithic applications is primarily performed by following an architectural pattern that favors a stateless application layer supported by a stateful database layer. All the stateless and stateful components communicate with each other via REST calls or message queues. Microservice architectures are well-known instances of this pattern.

At first sight, microservices are an obvious candidate for replacing monolithic applications and migrating to the cloud. Microservices offer code modularity, scalability, and development agility. However, microservices dismiss an important advantage that monolithic applications enjoyed for almost five decades: state management, failure management, and state consistency have been the responsibility of database systems. Today's microservice architectures depart from these DBMS amenities by intermingling state management, service messaging, and coordination with application logic. From the database community's point of view, the microservice architectural pattern resembles the situation long ago [27], when developers were implementing ad-hoc application-level transactions to ensure database consistency. Worse, managing communication and state in a distributed cloud environment increases complexity.

 15th Annual Conference on Innovative Data Systems Research (CIDR '25). January 19-22, Amsterdam, The Netherlands

For instance, in a shopping cart application, to complete a checkout, we first need to ensure enough stock of the selected products and then receive the corresponding payment before shipping the products. In the microservice paradigm, each service (Cart, Stock, Payment) has its own API, database, and application logic and communicates with other services through API calls. The main issue with microservices is that both atomicity (i.e., update stock *and* get paid for an order or cancel both actions) and state consistency across workflows (i.e., the stock counts should reflect the successfully paid orders) have to be solved in application code.

Similarly, Function-as-a-Service (FaaS) embraces the same general architecture pattern as microservices: stateless application, external database, and communication via messages. An orchestration layer on top of FaaS enables the composition of complex workflows to build service-oriented applications. However, orchestrators [14–16] solve only part of the problem, namely the atomicity of a workflow's execution. Moreover, achieving atomicity typically requires developers to handcraft compensating actions to roll back changes correctly using the SAGA pattern [17]. To address these concerns, a line of research [19, 38] proposes FaaS systems for workflow orchestration with transactional guarantees at the expense of performance and high-level programming primitives. For applications requiring low-latency transaction execution and state consistency across services [21], important challenges remain open.

In this paper, we first identify the limitations and shortcomings of microservice-like architectures for implementing transactional applications and then motivate the need for dedicated runtimes in order to support transactional cloud applications. We argue that to remove transaction- and failure-handling code from the application level, we need to address complex orchestration, service calls, and state management in a *holistic* manner at the system level, i.e., via a dedicated runtime. During the last years, we have been developing such a runtime for transactional applications called Styx [28]. Styx automatically partitions state, parallelizes function execution, and enables arbitrary transactional workflows to be executed with low latency. Most importantly, Styx's programming model [29] allows for application development that resembles a single-node application/monolith while transparently handling the serializable execution of massively parallel workflows in the cloud.

Our work is in line with recent research, such as Orleans [3], DBOS [24], Hydroflow [9], and SSMSs [25]. Contrary to these systems, our work adopts the streaming dataflow execution model while exposing an object-oriented/actor-like programming model on top [29] and guarantees serializability *across* services. To summarize, we make the following contributions:

- We analyze the shortcomings of modern cloud applications by exemplifying issues with current architectures and requirements for future systems.

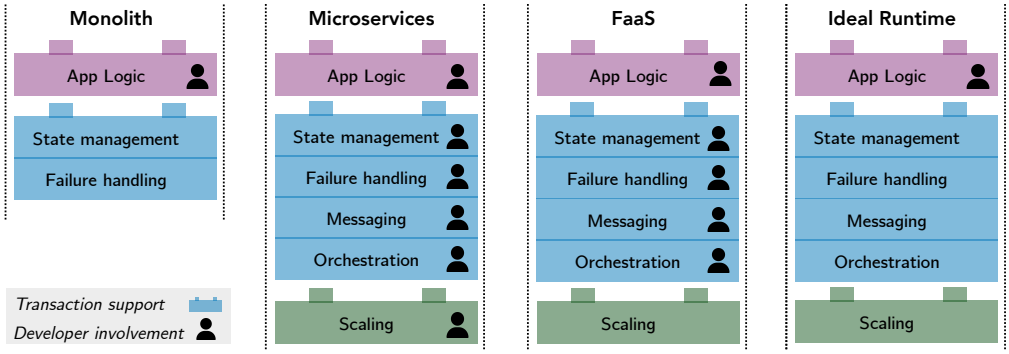


Figure 1: In monolithic applications, developers focused on application logic while a transactional database handled state management and failure recovery. In distributed cloud applications, development involves more challenges (e.g., failures, exactly-once messaging, and orchestration for atomicity and scalability). The ideal runtime should offer the same state consistency and ease of programming as monoliths, with improved scalability, without developer involvement.

- We provide arguments on the suitability of the stateful streaming dataflow paradigm for transactional cloud applications.
- We introduce a novel approach that brings together ideas from deterministic databases, dataflow systems, and serverless architectures alongside preliminary experiment results.
- We blueprint our future research directions in the area of transactional cloud application runtimes.

2 FROM MONOLITHS TO MICROSERVICES

As illustrated in Figure 1, developers in monolithic architectures were primarily responsible for the application logic. At the same time, with the adoption of microservices, they need to deal with messaging and failures (Section 2.1), state management and orchestration (Section 2.2), and scaling techniques (Section 2.3). Interestingly, in Figure 2, we observe that these aspects are not orthogonal. The conversion to a partitioned, event-driven architecture (Figure 2b to Figure 2c) requires state migration, coordination, and fault-tolerance.

Figure 2 depicts the process of breaking down a monolithic application (Figure 2a) into three microservices, each with their database (Figure 2b). In the microservice architecture, direct access to a single database and DBMS-based transactions are no longer possible. Instead, the microservices split functionality and maintain their database. Each service's database is partitioned to scale out, as shown in (Figure 2c). REST API calls are also transformed into messages that asynchronously trigger those calls.

Microservices Implement Dataflows. A critical observation is that the architecture depicted in Figure 2c closely resembles a streaming dataflow graph with the partitioned state co-located with the application logic. While we elaborate on this in Section 3.2, in short, this architectural pattern is the same pattern that is followed by streaming dataflow systems such as Apache Flink [6] and Spark Streaming [37].

2.1 Messaging, Idempotency & Consistency

Traditional monoliths achieved atomicity of workflow execution (e.g., a shopping cart checkout) by combining the state mutations of different subsystems (cart, payment, stock) in the same transaction. If the transaction fails, the database rolls back to the previous state, and the application retries to execute the checkout anew.

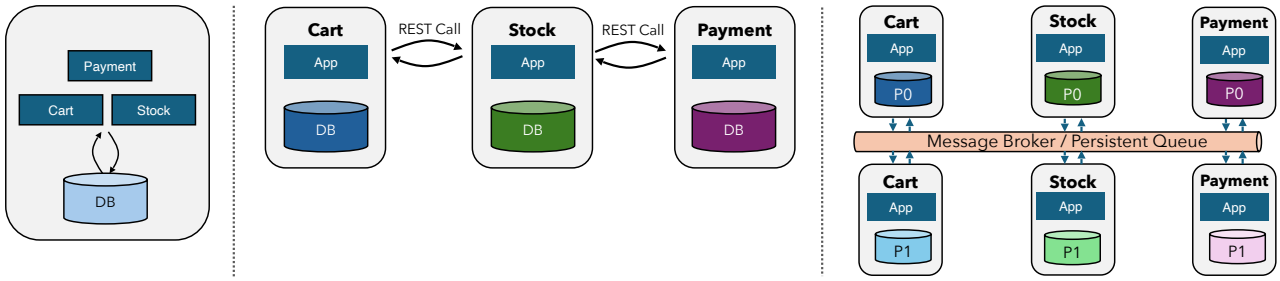
Idempotency in Services. To achieve the same effect, a stateful service or function must be idempotent, meaning that calling the service multiple times should have the same effect on the global state of an application as calling it exactly once. Considering that various issues can arise when two services communicate (such as network failures, rescaling, or service restarts), currently ensuring idempotency works as follows: the sender service generates an `idempotency-key`[1] that is persisted in the state of the sender, right before the call is performed. Suppose the sender sends a message twice (e.g., because of an intermittent network issue or a failure). In that case, the `idempotency-key` has to be recognized and safely ignored by the receiver service. It is important to note here that idempotency cannot be achieved without *persisting* the idempotency-key to durable storage (e.g., a database) in the *same local transaction* as the one that mutates the state of the receiver. At the moment, `idempotency-keys` are managed by the developers, adding to the complexity of developing cloud applications.

2.2 Transactions & Orchestration

Serializability in Services. Multiple works advocate that serializable guarantees are preferred [8, 23]. This is also reflected in the offered isolation levels of the post-NoSQL systems such as Google's Spanner [11] and, more recently, CockroachDB [35]: they all provide serializability. Serializability has been highly important in monolithic applications, but in distributed service deployments, it is virtually impossible to reason about correctness in case of state inconsistencies [23]. Transactional service architectures have to deal with message delivery guarantees.

SAGAs and Two-Phase Commit. Popular solutions to this challenge, known for years, involve the Saga pattern and two-phase commit protocols orchestrated by a transaction coordinator implementing XA transactions [33]. However, both of them present

[1] https://datatracker.ietf.org/doc/html/draft-idempotency-header-00



a) Monolith b) Microservice c) Partitioned, event-driven architecture

Figure 2: Three-step process of converting a monolith to a scalable, low-latency service architecture.

significant drawbacks. Implementing the Saga pattern involves managing the execution of compensating actions to reverse the partial state effects of a failed workflow while the offered consistency level is eventual. Alternatively, 2PC protocols coupled with two-phase locking provide atomicity and isolation at the expense of blocking the progress of service orchestrations involved in a transaction. We need a new way to architect cloud applications with support for transactional workflows that span multiple components of an application.

Orchestrators. Currently, several commercial orchestrators are available for executing SAGAs. Those orchestrators ensure atomicity only: they make sure that a given sequence of service calls eventually comes to completion. While we do see the value of orchestrators for analytics applications (e.g., as Apache Airflow [2], AWS Step Functions [15]), orchestrators are not suitable for transactional applications, as they are all *oblivious* of the state of the functions/services that they are orchestrating.

2.3 Application (Re-)Scaling

Scaling microservices requires scaling the stateless business logic and the state management system that serves the stateless part of an application. Scaling stateless services is relatively straightforward: one needs to rescale the application logic instance, assuming that the database behind the stateless instance can handle the new load. However, when optimizing for latency, the database is partitioned and preferably co-located with the application logic. In that case, rescaling an application becomes a hurdle: the database has to migrate state and possibly keep replicas. Soon enough, application developers re-implement some version of database state migration and rescaling [30] protocol.

While current FaaS cloud offerings do allow for stateless functions to scale on demand, they still provide no transaction management primitives that take into account service orchestrations and state consistency issues during the rescaling process. An ideal runtime should be able to perform the rescaling of applications without forcing operations teams and developers to perform rescaling by hand while keeping the state across services transactionally consistent.

3 STREAMING DATAFLOWS TO THE RESCUE

In this section, we highlight the key aspects and advantages of streaming dataflow systems design and argue that they can be extended to encapsulate the primitives required for executing transactional cloud applications consistently and efficiently. Moreover, we argue that combining deterministic databases and dataflow systems can create a runtime that ensures atomicity, consistency, and scalability. Finally, we show how deterministic databases can be extended for SFaaS, where transaction boundaries are unknown, unlike online transaction processing (OLTP).

3.1 Dataflows as an Architectural Abstraction

Stateful dataflows is the execution model implemented by virtually all modern stream processors [13]. Streaming systems owe their wide adoption in the last decade to a set of key system design aspects: exactly-once processing, consistent fault tolerance, co-location of state and compute, and data-parallel scale-out architecture. We elaborate on these characteristics below.

Exactly-once Processing. Message-delivery guarantees are fundamentally hard to deal with in the general case, with the root of the problem being the well-known Byzantine Generals problem. However, in the closed world of dataflow systems, exactly-once processing is possible [5, 6]. In principle, to achieve exactly-once processing, the processing layer records the outcome of each message's state effects, and the networking layer ensures the delivery of messages in FIFO order, while the fault tolerance layer guarantees that no message that is already reflected in the state will be processed again. Note that the guarantee of exactly-once processing facilitates programming substantially. The APIs of popular streaming dataflow systems, such as Apache Flink, require no error management code (e.g., message retries or duplicate elimination with `idempotency-keys`).

Fault Tolerance. Exactly-once processing extends to the system's fault tolerance approach. The two can be gracefully combined using Chandy-Lamport's distributed snapshot protocol [7] adapted for streaming systems [5, 31]. The approach involves periodically circulating special messages called checkpoint markers into the streaming dataflow system, instructing its operators to snapshot their state. Because checkpoint markers coexist with common data-related messages on the same channel, they enforce a global order that creates a consistent cut of the system's state. In case of a failure, the system can automatically roll back to the latest checkpoint of its distributed state and resume processing from that point, assuming the input is delivered from a replayable source, such as Apache

Kafka [20]. This fault tolerance approach ensures that the system's state remains consistent under failures.

Co-location of State with Compute. Streaming dataflow systems have demonstrated their capacity to process millions of events per second [6]. One main design decision that enables this level of sustainable performance is that the system's operators maintain the state of their computations in their local memory space. The state is periodically snapshotted to persistent storage, securing the progress of continuous computations against failures. Notably, this coarse-grained approach bears a low overhead to the system's regular operation.

Data-parallel Scale-out Architecture. Continuing from the previous point, the system's architecture enables high-throughput at scale. Each operator in the logical dataflow graph is instantiated as several operator instances deployed in distributed nodes. Each instance holds a partition of the operator's state, enabling input data to be distributed and processed across the instances in parallel.

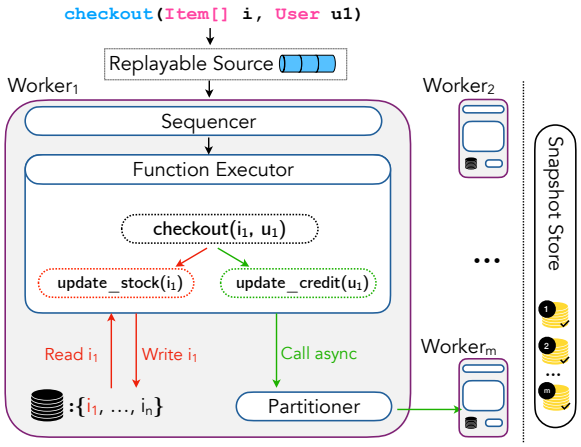


Figure 3: Stateful-Function execution in Styx.

3.2 Dataflows for Transactional Applications

The aforementioned advantages of streaming dataflow systems do not apply to transactional cloud applications. To begin with, typical transactional workloads in the cloud manifest as workflows of functions that arbitrarily call one another. This computation pattern is markedly different from analytics functions that populate the operators of streaming systems. Second, streaming dataflow systems lack support for transactions as prescribed in the database literature [27]. Finally, the development of workflows of functions entails a programming model that can convey transactional semantics, form workflows, and support custom business logic. This programming model departs from the typical way of programming stream-processing jobs as chained functional transformations.

Dataflows for Arbitrary-Workflow Execution. The prime use case for dataflow systems nowadays is streaming analytics, which typically involves executing a chain of standalone functions. By comparison, transactional cloud applications involve arbitrary workflows of functions calling each other. To enable the execution of arbitrary workflows in a dataflow system, we connect operators at the system level such that an operator can directly invoke a computation in another operator. In addition, we allow such nested computations to be executed in parallel. Finally, we devised an approach for identifying the transaction boundaries of a workflow, which we briefly describe next.

Deterministic transactions. Deterministic transactional protocols have two properties that make them coexist harmoniously with dataflow systems. First, given a set of sequenced transactions, a deterministic database [1, 36] will end up in the same final state with serializable guarantees despite node failures and possible concurrency issues. This property is essential because it allows a deterministic transactional protocol to align with a dataflow system without changing the stream processor's checkpointing mechanism.

Second, unlike 2PC, which requires rollbacks in case of failures, deterministic database protocols [26, 36] are "forward-only": once the locking order [36] or read/write set [26] of a batch of transactions has been determined, the transactions will be executed and reflected on the database state, without the need to rollback changes. This alignment between deterministic databases and the dataflow execution model is the primary motivation to support a deterministic transaction protocol on top of a dataflow system.

Still, supporting deterministic transactions in a streaming dataflow system is not trivial and poses two main challenges that we address in our prototype system presented in Section 4. The first challenge is how to determine transaction boundaries. This is not required in deterministic databases where each transaction is encapsulated in a single-threaded function that can execute remote reads and writes from other partitions [26, 36]. In SFaaS, however, arbitrary function calls to remote partitions are common because they enable developers to take advantage of both the separation of concerns principle that is widely applied in microservice architectures [21], as well as code modularity. Therefore, to determine a transactional workflow's boundaries, we introduce an accounting scheme for function calls nested inside a workflow. The scheme, which also supports calls to remote operators and cycles, signals the termination of a workflow's execution once all function calls complete.

The second challenge is deciding when to commit to durable storage and reply to users. Traditionally, a transactional system can respond to a client only when *i*) the requested transaction has been committed to a persistent, durable state or *ii*) the write-ahead log is flushed and replicated. Within the scope of a dataflow system, this would require completing a snapshot, leading to prohibitive latency. However, a deterministic transactional protocol executes an agreed-upon sequence of transactions among the workers; after a failure, the system would run the same transactions with exactly the same effects. This determinism allows for early commit replies: the client can receive a reply before a persistent snapshot is stored.

Programming Models. Currently, dataflow systems are only programmable through functional-programming style dataflow APIs: a given cloud application needs to be rewritten by developers to match the event-driven dataflow paradigm. Although it is possible to rewrite many applications in this paradigm, it takes a considerable amount of programmer training and effort to do so. Therefore, we have introduced an object-oriented programming abstraction that encapsulates functions into actor-like entities. We present the programming model as a whole in Section 4.1. We argue that this

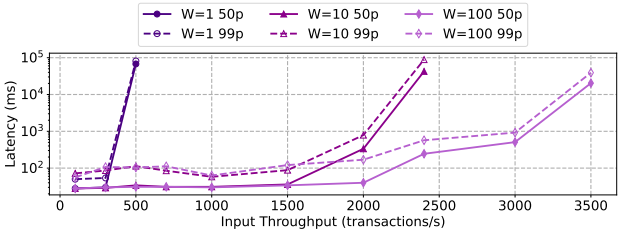


Figure 4: Median and 99p Latency against throughput using the TPC-C workload on Styx with 1, 10, and 100 warehouses.

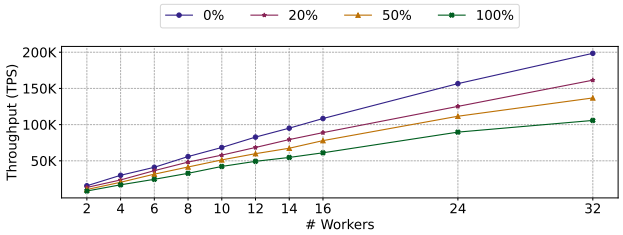


Figure 5: Scalability of Styx on YCSB with varying percentages of multi-partition transactions.

programming model is suitable for developing transactional cloud applications like microservices.

4 THE STATEFLOW/STYX APPROACH

Styx [28] is a transactional distributed dataflow system that executes workflows of stateful functions with serializable guarantees. Styx adopts Stateflow [29] as a higher-level programming abstraction, enabling users to code in a pure object-oriented style without state management or fault tolerance considerations. In this section, we describe the programming model (Section 4.1) and underlying system (Section 4.2).

4.1 Programming Model

The Stateflow/Styx framework provides developers with two levels of abstraction: a high-level actor-like programming interface based on Stateflow [29] and a lower-level dataflow API [28].

High-level. Users can code transactional cloud applications in Python object-oriented code where an entity is an object with a unique key and class functions that mutate the entity's state (similar to actor programming). Additionally, when an entity calls a function of another entity, Stateflow automatically creates an edge in the dataflow graph. We describe Stateflow's workings and how it uses continuation-passing style programming to transform calls between different entities into a distributed dataflow graph in [29].

Low-level. Styx follows the operator API of dataflow systems (e.g., Apache Flink [6]). In Styx, a streaming operator can hold multiple entities based on a partitioning scheme, the functions that act upon the operator as a whole (allowing for range queries), or the entities themselves (point queries). To communicate across operators, developers can call remote operator functions using Styx's API.

4.2 The Styx Runtime

Styx [28] (Figure 3) employs a typical worker/coordinator architecture and is complemented by a messaging system, such as Apache Kafka, that propagates input to Styx, including the replay of unprocessed messages following a failure. The coordinator's responsibilities are to deploy a user-defined dataflow graph to the workers, monitor the cluster's health while collecting useful metrics, and trigger the fault tolerance pipeline in case of failure.

The workers are responsible for a subset of the dataflow graph's operator state partitions, which are 1-to-1 aligned with the partitions of the replayable input source, say Apache Kafka. First, each worker ingests client requests through Kafka and sequences them (Styx uses a non-replicated sequencer partitioned per worker). Then, it receives a batch of transactions from the sequencer and executes them as coroutines in a single CPU to increase efficiency. To execute transactions in a deterministic fashion, Styx extends a deterministic transactional protocol similar to Calvin [36] and Aria [26]. Determinism is required by the dataflow snapshotting mechanism to guarantee the same state mutations after a replay in case of failure. Transactions are executed in parallel across workers, and nested function calls are transparently scheduled for execution by local or remote operators. Finally, Styx's acknowledgment-sharing scheme signals the end of a transaction's execution.

Fault Tolerance. To recover from failures, Styx relies on a replayable input source to perform deterministic message replay based on recorded offsets. This design ensures that the sequencer will re-create the same transaction sequence post-recovery and enables early replies (before the state commits to durable storage). Finally, Styx utilizes a blob store to persist incremental snapshots of worker states.

5 PRELIMINARY RESULTS

We conducted a series of preliminary experiments to evaluate the performance of our proposed approach. Our initial results indicate that our system can support large-scale cloud applications with high performance and near-linear scalability.

TPC-C. One of the most popular transactional benchmarks targeting OLTP systems is TPC-C [22]. To use it in our evaluation, we had to transform it into an event-driven microservice workload. Each table is now a microservice (or entity), leading to 12 microservices (9 for the tables, 2 to hold the transaction metadata, and 1 for the customer index). In Figure 4, we showcase results while running the NewOrder and Payment transactions with equal probability, a varying amount of warehouses (1, 10, 100), and 100 Styx workers (1 CPU/2GB RAM per worker). The rewrite required splitting the NewOrder transaction into 20-50 function calls (one call for each item in the NewOrder transaction) and the Payment transaction into 8 function calls. TPC-C scales in size/partitions by increasing the number of warehouses represented in the benchmark. While a single warehouse represents a skewed workload (all transactions will hit the same warehouse), increasing the number of warehouses

decreases contention, allowing for higher throughput and lower latency.

In Figure 4, we observe that Styx's performance improves as we increase the input throughput for different numbers of warehouses, reaching up to 3K TPS (or 180K tpmC) with sub-second 99^{th} percentile latency (100 warehouses). The latency is measured end-to-end (i.e., from the client and back), and the clients are in the same cluster in our experiments.

Scalability. In this experiment, we test the scalability of Styx by increasing the number of Styx workers (1 CPU each). Each worker has a state of 1 million keys. We measure the maximum throughput on YCSB [10]. The goal is to calculate the speedup of operations as the input throughput and number of workers scale together. In addition, we control the percentage of multi-partition transactions in the workload, i.e., transactions that span across workers. In Figure 5, we observe that Styx retains near-linear scalability in all settings. Finally, Styx displays the expected behavior as multi-partition transactions increase. The codebase and experiments of Styx can be found here: https://github.com/delftdata/styx

6 RELATED WORK

Our system shares motivation with projects such as Hydroflow [9] and DBOS [24]. DBOS takes a DB-centric approach where functions can be translated to stored procedures within a database (co-location of state and processing) or in the server where the state needs to be transferred, and workflows form a database transaction with ACID guarantees. Hydroflow, at its present state, does not support transactional end-to-end workflows and focuses primarily on cloud-native stream processing for analytics. Cloudburst [34] provides causal consistency guarantees within a single Directed Acyclic Graph (DAG) workflow. Netherite [4] offers exactly-once execution guarantees and a high-level programming model, though it does not ensure transactional serializability across functions. Orleans [3] introduces virtual actors decoupling applications from the underlying architecture but does not guarantee exactly-once message delivery. Finally, transactional SFaaS paradigms with serializability guarantees (Beldi [38], Boki [19], and T-Statefun [12]) do support transactional end-to-end workflows but suffer in performance and fail to decouple the user code from their transactional primitives.

7 THE ROAD AHEAD

Our work aims at simplifying the development of transactional cloud applications by providing transactional support, scalability, and fault tolerance. There are two important milestones on the road to achieving this goal: i) make Styx serverless, i.e., scale up and down to zero without any input from a cluster manager (note that from a programmer's perspective, the API is already serverless) and ii) enable Styx to interact deterministically with external systems (e.g., another microservice deployment).

Serverless Runtime. The first step towards making a dataflow system serverless by scaling up and down is state migration, which is thoroughly explored in both dataflow [18] and transactional systems [30]. However, since Styx is a combination of both, it leads to a fundamental challenge that none of the two breeds supports: how to maintain transactional guarantees during state migration in the presence of failures.

Support Non-Deterministic Operations. In principle, functions may encapsulate logic that makes the outcome of their execution non-deterministic. Examples of non-deterministic operations are calls to external systems and the use of random number generators or time-related utilities. Styx, like state-of-the-art SFaaS systems [19, 38], currently supports deterministic functions. That said, it is possible to embrace non-deterministic functions in Styx by tracking, recording, and persisting non-deterministic logic contained in them following the approach of Clonos [32].

ACKNOWLEDGMENTS

This publication is part of project number 19708 of the Vidi research program, partly financed by the Dutch Research Council (NWO).

REFERENCES

[1] Daniel J Abadi and Jose M Faleiro. 2018. An overview of deterministic database systems. *Commun. ACM* 61, 9 (2018), 78–88.
[2] Apache Airflow. 2015. https://airflow.apache.org/.
[3] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. Orleans: Distributed virtual actors for programmability and scalability. *MSRTR2014* 41 (2014).
[4] 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. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1591–1604.
[5] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.
[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).
[7] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.
[8] 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.*
[9] Alvin Cheung, Natacha Crooks, Joseph M Hellerstein, and Mae Milano. 2021. New directions in cloud programming. In *11th Annual Conference on Innovative Data Systems Research (CIDR '21), January 10-13, 2021, Chaminade, USA.*
[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing.* 143–154.
[11] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.
[12] 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
[13] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2024. A survey on the evolution of stream processing systems. *The VLDB Journal* 33, 2 (2024), 507–541.
[14] Azure Durable Functions. 2018. https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.
[15] AWS Step Functions. 2016. https://aws.amazon.com/step-functions/.
[16] Google Cloud Run functions. 2024. https://cloud.google.com/functions.
[17] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. *ACM Sigmod Record* 16, 3 (1987), 249–259.
[18] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. 2019. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1002–1015.
[19] 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.* 691–707.

[20] 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.

[21] 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 (Sept. 2021), 3348–3361. https://doi.org/10.14778/3484224.3484232

[22] Scott T Leutenegger and Daniel Dias. 1993. A modeling study of the TPC-C benchmark. *ACM Sigmod Record* 22, 2 (1993), 22–31.

[23] Qian Li, Peter Kraft, Michael Cafarella, Çağatay Demiralp, Goetz Graefe, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2023. Transactions Make Debugging Easy. In *13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.*

[24] Qian Li, Peter Kraft, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Jason Li, Michael J Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, et al. 2022. A Progress Report on DBOS: A Database-oriented Operating System.. In *CIDR.*

[25] Tianyu Li, Badrish Chandramouli, Sebastian Burckhardt, and Samuel Madden. 2024. Serverless State Management Systems.. In *CIDR.*

[26] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proc. VLDB Endow.* (2020).

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

[28] Kyriakos Psarakis, George Siachamis, George Christodoulou, Marios Fragkoulis, and Asterios Katsifodimos. 2024. Styx: Transactional Stateful Functions on Streaming Dataflows. arXiv:2312.06893 [cs.DC] https://arxiv.org/abs/2312.06893

[29] Kyriakos Psarakis, Wouter Zorgdrager, Marios Fragkoulis, Guido Salvaneschi, and Asterios Katsifodimos. 2024. Stateful entities: object-oriented cloud applications as distributed dataflows. *EDBT* (2024).

[30] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456.

[31] George Siachamis, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen, Paris Carbone, and Asterios Katsifodimos. 2024. CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows. In *2024 IEEE 40th International Conference on Data Engineering (ICDE).* 4030–4043. https://doi.org/10.1109/ICDE60146.2024.00309

[32] Pedro F Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. 2021. Clonos: Consistent causal recovery for highly-available streaming dataflows. In *Proceedings of the 2021 International Conference on Management of Data.* 1637–1650.

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

[34] 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. *Proceedings of the VLDB Endowment* 13, 11 (2020).

[35] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data.* 1493–1509.

[36] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12).* Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[37] 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.

[38] 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).* 1187–1204.