Styx: Transactional Stateful Functions on Streaming Dataflows

KYRIAKOS PSARAKIS, Delft University of Technology, The Netherlands
GEORGE CHRISTODOULOU, Delft University of Technology, The Netherlands
GEORGE SIACHAMIS*, Inria & Institut Polytechnique de Paris, France
MARIOS FRAGKOULIS, Delft University of Technology, The Netherlands
ASTERIOS KATSIFODIMOS, Delft University of Technology, The Netherlands

Developing stateful cloud applications, such as low-latency workflows and microservices with strict consistency requirements, remains arduous for programmers. The Stateful Functions-as-a-Service (SFaaS) paradigm aims to serve these use cases. However, existing approaches provide weak transactional guarantees or perform expensive external state accesses requiring inefficient transactional protocols that increase execution latency. In this paper, we present Styx, a novel dataflow-based SFaaS runtime that executes serializable transactions consisting of stateful functions that form arbitrary call-graphs with exactly-once guarantees. Styx extends a deterministic transactional protocol by contributing: i) a function acknowledgment scheme to determine transaction boundaries required in SFaaS workloads, ii) a function-execution caching mechanism, and iii) an early commit-reply mechanism that substantially reduces transaction execution latency. Experiments with the YCSB, TPC-C, and Deathstar benchmarks show that Styx outperforms state-of-the-art approaches by achieving at least one order of magnitude higher throughput while exhibiting near-linear scalability and low latency.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Information systems → Data management systems**.

Additional Key Words and Phrases: Streaming Dataflows; Serializable Deterministic Transactions; Stateful Functions

ACM Reference Format:
Kyriakos Psarakis, George Christodoulou, George Siachamis, Marios Fragkoulis, and Asterios Katsifodimos. 2025. Styx: Transactional Stateful Functions on Streaming Dataflows. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 226 (June 2025), 28 pages. https://doi.org/10.1145/3725363

1 Introduction

Despite the commercial offerings of the Functions-as-a-Service (FaaS) cloud service model, its suitability for low-latency stateful applications with strict consistency requirements, such as payment processing, reservation systems, inventory keeping, and low-latency business workflows, is quite limited. The reason behind this unsuitability is that current FaaS solutions are stateless,

*Work done while at Delft University of Technology

Authors' Contact Information: Kyriakos Psarakis, Delft University of Technology, Delft, The Netherlands, k.psarakis@tudelft.nl; George Christodoulou, Delft University of Technology, Delft, The Netherlands, g.c.christodoulou@tudelft.nl; George Siachamis, Inria & Institut Polytechnique de Paris, Paris, France, georgios.siachamis@inria.fr; Marios Fragkoulis, Delft University of Technology, Delft, The Netherlands, m.fragkoulis@tudelft.nl; Asterios Katsifodimos, Delft University of Technology, Delft, The Netherlands, a.katsifodimos@tudelft.nl.

ACM 2836-6573/2025/6-ART226
https://doi.org/10.1145/3725363

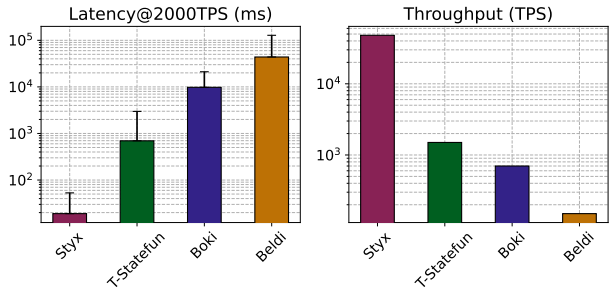


Fig. 1. Styx outperforms the SotA by at least one order of magnitude in transactional workloads (§8). The figure shows median (bar)/99p (whisker) latency and throughput. For the latency plot, the input throughput is 2000 transactions per second (TPS), and for the throughput plot, we report the throughput that the systems achieve at subsecond latency.

relying on external, fault-tolerant data stores (blob stores or databases) for state management. In addition, while multiple frameworks can perform workflow execution (e.g., AWS Step Functions [2], Azure Logic Apps [44]), they do not provide primitives for *transactional* execution of such applications. As a result, distributed applications (e.g., microservice architectures) suffer from serious consistency issues when the responsibility of transaction execution is left to developers [10, 34, 59].

In line with recent research [12, 28, 29, 55, 56, 63], we agree that for FaaS offerings to become mainstream, they should include state management support for stateful functions according to the Stateful Functions-as-a-Service (SFaaS) paradigm. In addition, we argue that a suitable runtime for executing workflows of stateful functions should also provide *i*) end-to-end serializable transactional guarantees across multiple functions, *ii*) low-latency and high-throughput execution, and *iii*) a high-level programming model, devoid of low-level primitives for locking and transaction coordination. To the best of our knowledge, no existing approach addresses all these requirements together.

The state-of-the-art transactional SFaaS with serializable guarantees, Boki [28], Beldi [63], and T-Statefun [12] do support transactional end-to-end workflows but induce high commit latency and low throughput. The main reason behind their inefficiency is the separation of state storage and function logic, as well as the use of locking and Two-Phase Commit (2PC) [23] to coordinate and ensure the atomicity of cross-function transactions.

This paper proposes Styx, a novel dataflow-based runtime for SFaaS. Styx ensures that each transaction's state mutations will be reflected once in the system's state, even under failures, retries, or other potential disruptions (known as exactly-once processing). Additionally, Styx can execute arbitrary function orchestrations with end-to-end serializability guarantees, leveraging concepts from deterministic databases to avoid costly 2PCs.

Our work stems from two important observations. First, modern streaming dataflow systems such as Apache Flink [8] guarantee exactly-once processing [7, 8, 53] by transparently handling failures. A limitation of those streaming systems is that they cannot execute general cloud applications such as microservices or guarantee transactional SFaaS orchestrations. Second, deterministic database protocols [42, 61] that can avoid expensive 2PC invocations have not been designed for complex function orchestrations and arbitrary call-graphs. For the needs of transactional SFaaS, Styx leverages a deterministic transactional protocol, enabling early commit-replies to clients (i.e., before a snapshot is committed to persistent storage).

Our work is in line with recent proposals in the area, such as DBOS [54], Hydro [11], and SSMSs [39]. Unlike these systems, our work adopts the streaming dataflow execution model and guarantees serializability *across* functions. As shown in Figure 1, Styx achieves one order of magnitude lower median latency, two orders of magnitude lower 99p latency at 2000 transactions/sec, and one order of magnitude higher throughput compared to state of the art (SotA) serializable SFaaS systems [12, 28, 63].

In short, this paper makes the following contributions:
– Styx combines deterministic transactions with dataflows and overcomes the challenges that arise from this design choice (§2).
– Styx enables high-level SFaaS programming models that abstract away transaction and failure management code (§3). Styx does so, by guaranteeing exactly-once processing (§4) and transactional serializability across arbitrary function calls (§5 and §6).
– Styx extends the concept of deterministic databases to support arbitrary workflows of stateful functions, contributing a novel acknowledgment scheme (§5.3) to track function completion efficiently, as well as a function-execution caching mechanism (§6.3) to speed up function re-executions.
– Styx's deterministic execution enables early commit-replies: transactions can be reported as committed even before a snapshot of executed transactions is committed to durable storage (§6.4).
– Styx outperforms the state-of-the-art [12, 28, 63] by at least one order of magnitude higher throughput in all tested workloads while achieving lower latency and near-linear scalability (§8).

Styx is available at: https://github.com/delftdata/styx

2 Motivation

In this section, we analyze the specifics of streaming dataflow systems design and argue that they can be extended to encapsulate the primitives required for consistently and efficiently executing workflows of stateful functions. Our work is based on a key observation: the architecture of high-performance cloud services closely resembles a parallel dataflow graph, where the state is partitioned and co-located with the application logic [49]. Additionally, as we detail in §2.2, there is a synergy between deterministic transactions and dataflow systems. Such a combination can offer state consistency and ease of programming as monolithic solutions did in the past, while improving scalability and eliminating developer involvement. Finally, we show how deterministic transactions can be extended for SFaaS, where transaction boundaries are unknown, unlike online transaction processing (OLTP).

2.1 Dataflows for Stateful Functions

Stateful dataflows is the execution model implemented by virtually all modern stream processors [8, 46, 48]. Besides being a great fit for parallel, data-intensive computations, stateful dataflows are the primary abstraction supporting workflow managers such as Apache Airflow [18], AWS Step Functions [2], and Azure's Durable Functions [6]. In the following, we present the primary motivation behind using stateful dataflows to build a suitable runtime for orchestrating general-purpose cloud applications.

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 [35]. However, in the closed world of dataflow systems, exactly-once processing is possible [7, 8, 53]. As a matter of fact, 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 IDs).

Co-Location of State and Function. The primary reason streaming dataflow systems can sustain millions of events per second [8, 21] is that their state is partitioned across operators that operate on local state. While the structure of current Cloud offerings favors the disaggregation of storage and computation, we argue that co-locating state and computation enables high performance and can also be adopted by modern SFaaS runtimes, as opposed to using external databases for state storage.

Coarse-Grained Fault Tolerance. To ensure atomicity at the level of workflow execution, existing SFaaS systems perform fine-grained fault tolerance [28, 63]; each function execution is logged and persisted in a shared log before the next function is called. This requires a round-trip to the logging mechanism for each function call, which adds significant latency to function execution. Instead of logging each function execution, streaming dataflow systems [7, 9, 52] opt for a coarse-grained fault tolerance mechanism based on asynchronous snapshots, reducing this overhead.

2.2 Determinism & Transactions

Given a set of database partitions and a set of transactions, a deterministic database [1, 61] will end up in the same final state despite node failures and possible concurrency issues. Traditional database systems offer *serializable* guarantees, allowing multiple transactions to execute concurrently and ensure that the database state will be equivalent to the state of one serial transaction execution. Deterministic databases guarantee not only serializability but also that a given set of transactions will have exactly the same effect on the database state despite transaction re-execution. This guarantee has important implications [1] that have not been leveraged by SFaaS systems thus far.

Deterministic Transactions on Streaming Dataflows. Unlike 2PC, which requires rollbacks in case of failures, deterministic database protocols [42, 61] are "forward-only": once the locking order [61] or read/write set [42] of a batch of transactions has been determined, the transactions are going to be executed and reflected on the database state, without the need to rollback changes. This notion is in line with how dataflow systems operate: events flow through the dataflow graph, from sources to sinks, without stalls for coordination. This match between deterministic databases and the dataflow execution model is the primary motivation behind Styx's design choice to implement a deterministic transaction protocol on top of a dataflow system.

2.3 Challenges

Despite their success and widespread applicability, dataflow systems need to undergo multiple changes before they can be used for transactional stateful functions. In the following, we list challenges and open problems tackled in this work.

Programming Models. Dataflow systems at the moment are only programmable through functional programming-style dataflow APIs: a given cloud application has to be rewritten by programmers 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. We argue that dataflow systems would benefit from object-oriented or actor-like programming abstractions in order to be adopted for general cloud applications, such as microservices.

Support for Transactions. Transactions in the context of streaming dataflow systems typically refer to processing a set of input elements and their state updates with ACID guarantees [64]. Despite progress, critical challenges remain open, such as the performance overhead incurred by multi-partition transactions, as well as the need to block flows of data for locking and message re-ordering. In this work, we argue that in order to implement transactions in a streaming dataflow system, we need to "keep the data moving" [57] by avoiding disruptions in the natural flow of

data while tightly integrating transaction processing into the system's state management and fault tolerance protocols.

Deterministic OLTP and SFaaS. OLTP databases that use deterministic protocols like Calvin [42, 61, 66] either require each transaction's read/write set a priori or are extended to discover the read-write sets of a transaction by first executing it. Additionally, in both scenarios, deterministic protocols assume that a transaction is executed as a single-threaded function that can perform remote reads and writes from other partitions. In the case of SFaaS, arbitrary function calls enable programmers to take advantage of both the separation of concerns principle, which is widely applied in microservice architectures [34], as well as code modularity. Although deterministic database systems have been proven to perform exceptionally well [1], designing and implementing a deterministic transactional protocol for arbitrary workflows of stateful functions is non-trivial. Specifically, arbitrary function calls create complex call-graphs that need to be tracked in order to establish a transaction's boundaries before committing.

Dataflows for Arbitrary-Workflow Execution. The prime use case for dataflow systems nowadays is streaming analytics. However, general-purpose cloud applications have different workload requirements. Functions calling other functions and receiving responses introduce cycles in the dataflow graph. Such cycles can cause deadlocks and need to be dealt with [36].
In this work, we tackle these challenges and propose a dataflow system tailored to the needs of stateful functions with built-in support for deterministic transactions and a high-level programming model.

3 Programming Model

The programming model of Styx is based on Python and comprises operators that encapsulate partitioned mutable state and functions that operate on that. An example of the programming model of Styx is depicted in Figure 2.

3.1 Programming Model Notions

Stateful Entities. Similar to objects in object-oriented programming, `entities` in Styx are responsible for maintaining and mutating their own `state`. Moreover, when a given entity needs to update the state of another entity, it can do so via a function call. Each entity bears a unique and immutable `key`, similar to Actor references in Akka [40], with the difference that entity keys are application-dependent and contain no information related to their physical location. The dataflow runtime engine (§4) uses that key to route function calls to the right operator that accommodates that specific `entity`.

Functions. `functions` can mutate the state of an entity. By convention, the `context` is the first parameter of each function call. Functions are allowed to call other functions directly, and Styx supports both synchronous and asynchronous function calls. For instance, in lines 9-11 of Figure 2, the instantiated reservation entity will call asynchronously the function `'reserve_hotel'` of an entity with key `'hotel_id'` attached to the Hotel operator. Similarly, one can make a synchronous call that blocks waiting for results. In this case, Styx will block execution until the call returns. Depending on the use case, a mix of synchronous and asynchronous calls can be used. Asynchronous function calls, however, allow for further optimizations that Styx applies whenever possible, as we describe in §5 and §6.

Operators. Each `entity` directly maps to a dataflow operator (a vertex) in the dataflow graph. When an *event* enters the dataflow graph, it reaches the operator holding the *function code* of the given entity as well as the *state* of that entity. In short, a dataflow operator can execute all functions

```python
from styx import Operator
from deathstar.operators import Hotel, Flight

reservation_operator = Operator('reservation', n_partitions=4)

@reservation_operator.register
async def make_reservation(context, flight_id, htl_id, usr_id):

    context.call_async(operator=Hotel,
                       function_name='reserve_hotel',
                       key=htl_id)
    context.call_async(operator=Flight,
                       function_name='reserve_flight',
                       key=flight_id)

    reservation = {"fid":flight_id, "hid":htl_id, "uid":usr_id}
    await context.state.put(reservation)

    return "Reservation Successful"
```

Fig. 2. Deathstar's[20] Hotel/Flight reservation in Styx. From lines 9-14, the *reserve_hotel* and *reserve_flight* functions are invoked asynchronously. Finally, in lines 16-17, the reservation information is stored. In Styx, the transactional and fault tolerance logic are handled internally.

of a given entity and store the state of that entity. Since operators can be partitioned across multiple cluster nodes, each partition stores a set of stateful entities indexed by their unique `key`. When an entity's function is invoked (via an incoming event), the entity's state is retrieved from the local operator state. Then, the function is executed using the arguments found in the incoming event that triggered the call.

State & Namespacing. As mentioned before, each entity has access only to its own state. In Styx, the state is *namespaced* with respect to the entity it belongs to. For instance, a given key "`hotel53`" within the operator `Hotel` is represented as: `entities://Hotel/hotel53`. This way, a reference to a given key of a state object is unique and can be determined at runtime when operators are partitioned across workers. Programmers can store or retrieve `state` through the `context` object by invoking `context.put()` or `get()` (e.g., in Line 17 of Figure 2). Styx's `context` is similar to the context object used in other systems, such as Flink Statefun, AWS Lambda, and Azure Durable Functions.

Transactions. A transaction in Styx begins with a client request. The functions that are part of the transaction form a workflow that executes with serializable guarantees. Styx's programming model allows transaction aborts by raising an uncaught exception. In the example of Figure 2, if a hotel entity does not have enough availability when calling the *'reserve_hotel'* function, the *'make_reservation'* transaction should be aborted, alongside potential state mutations that the *'reserve_flight'* has made to a flight entity. In that case, the programmer has to raise an exception as follows:

```python
...
# Check if there are enough rooms available in the hotel
if available_rooms <= 0:
    raise NotEnoughSpace(f'No rooms in hotel: {context.key}')
...
```

The exception is caught by Styx, which automatically triggers the abort/rollback sequence of the transaction where the exception occurred and sends the user-defined exception message as a reply.

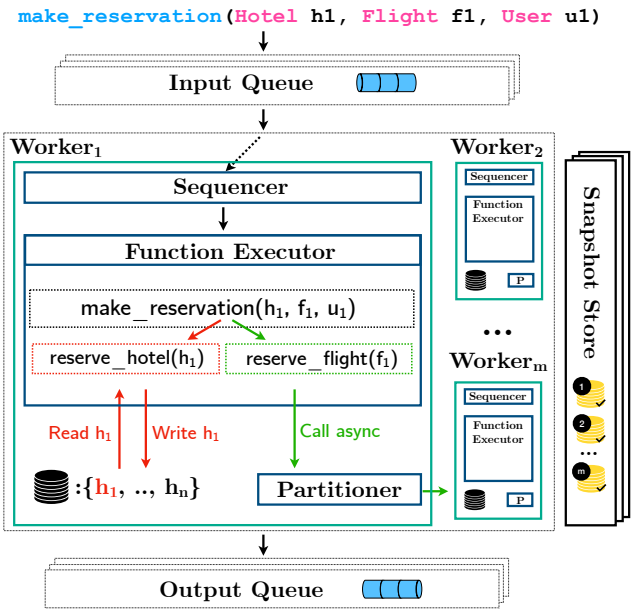


Fig. 3. Stateful-Function execution in Styx. In each worker, one coroutine manages the sequencing of incoming transactions while another coroutine handles their processing. In this example, transaction (`make_reservation`) consists of two functions: `reserve_hotel` and `reserve_flight`. A function can access local state (`reserve_hotel`) but also perform remote calls to different partitions (`reserve_flight`). This remote call uses the partitioner to locate the correct worker storing that partition.

Exactly-once Function Calling. Styx offers *exactly-once processing* guarantees: it reflects the state changes of a function call execution exactly-once. Thus, programmers do not need to "pollute" their application logic with consistency checks, state rollbacks, timeouts, retries, and idempotency [32, 34]. We detail this capability in §7.

4 Styx's Architecture

In this section, we describe the components (Figure 3) and the main design decisions of Styx.

4.1 Components

Coordinator. The coordinator manages and monitors Styx's workers, as well as the runtime state of the cluster (transactional metadata, dataflow state, partition locations, etc.). It also performs scheduling and health monitoring. Styx monitors the cluster's health using a heartbeat mechanism and initiates the fault-tolerance mechanism (§7) once a worker fails.

Worker. As depicted in Figure 3, the worker is the primary component of Styx, processing transactions, receiving or sending remote function calls, and managing state.

The worker consists of two primary coroutines. The first coroutine ingests messages for its assigned partitions from a durable queue and sequences them. The second coroutine receives a set of sequenced transactions and initiates the transaction processing. By utilizing the coroutine execution model, Styx increases its efficiency since the most significant latency factor is waiting for network or state-access calls. Coroutines allow for single-threaded concurrent execution, switching

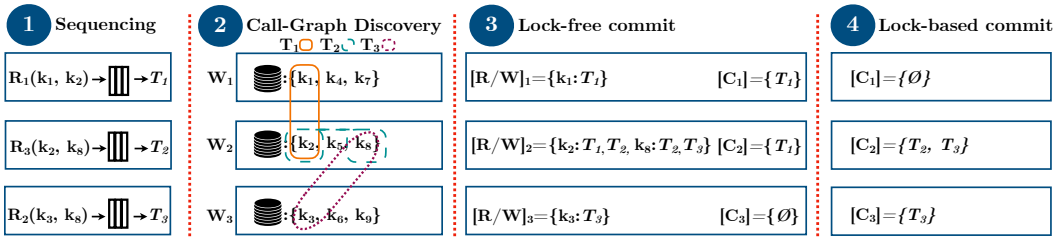


Fig. 4. The transaction execution pipeline in Styx is divided into 4 parts. First, each external request (R_i) is sequenced as a transaction and is assigned a unique id. Afterward, the transactions execute their application logic, accessing local keys and performing remote function calls. While a transaction executes, Styx tracks its accessed keys ($[R/W]_i$) and incrementally constructs its call-graph. Subsequently, Styx commits the transactions that do not participate in unresolved conflicts without having to perform locking. For example, we observe that workers W_1 and W_2 are capable to commit $C_1 = C_2 = \{T_1\}$ while T_1 interacts with the same keys as T_2; although it has the lowest id. In the final part, we commit all the transactions by resolving the conflicts with a lock-based mechanism ($C_2 = \{T_2, T_3\}$), $C_3 = \{T_3\}$).

between coroutines when one gets suspended during a network call, allowing others to make progress. Once the network call is completed, the suspended coroutine resumes processing.

Partitioning Stateful Entities Across Workers. Styx makes use of the entities' `key` to distribute those entities and their state across a number of workers. By default, each worker is assigned a set of keys using hash partitioning.

Input/Output Queue. For fault tolerance, Styx assumes a persistent input queue from which it receives requests from external systems (e.g., from a REST gateway API). Styx requires the input queue to be able to deterministically replay messages based on an offset when a failure occurs. As we detail in §7, the replayable input queue is necessary for Styx to produce the same sequence of transactions after the recovery is complete and to enable early commit-replies (§6.4). In the same way, Styx sends the result of a given transaction to an output queue from which an external system (e.g., the same REST gateway API) can receive it. Currently, Styx leverages Apache Kafka [33].

Durable Snapshot Store. Alongside the replayable queue, durable storage is necessary for storing the workers' snapshots. Currently, Styx uses Minio [45], an open-source blob store that follows the AWS S3 API, to store the incremental snapshots as binary data files.

4.2 Transaction Execution Pipeline

Styx employs an epoch-based transactional protocol that concurrently executes a batch of transactions in each epoch. A transaction may include multiple functions that, during runtime, form a call-graph of function invocations. Each function may mutate its entity's state, and the effects of function invocations are committed to the system state in a transactional manner. In Figure 3, once `make_reservation` enters the system, it is persisted and replicated by the input queue. Then, a worker ingests the call into its local sequencer that assigns a Transaction ID (TID) and processes all the encapsulated function calls as a single transaction. In the `make_reservation` case, the transaction consists of two functions: `reserve_hotel` and `reserve_flight`. For this example, let us assume that `reserve_hotel` is a local function call and `reserve_flight` runs on a remote worker. `reserve_hotel` will execute locally in an asynchronous fashion using coroutines and apply state changes. In contrast, `reserve_flight` will execute asynchronously on a remote worker, applying changes on the remote state.

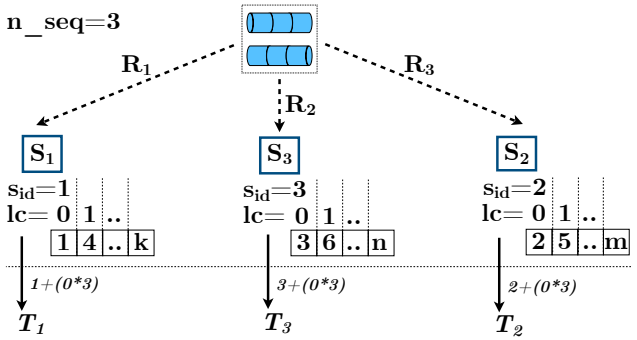


Fig. 5. Example of TID assignment in Styx with three sequencers. Their identifiers $\{1, 2, 3\}$ lead to the following sequences: $S_1 = \{1, 4, ..., k\}$, $S_2 = \{2, 5, ..., m\}$, $S_3 = \{3, 6, ..., n\}$ following the formula expressed in Equation (1).

5 Sequencing & Function Execution

The deterministic execution of functions with serializable guarantees requires a sequencing step that assigns a transaction ID (TID), which, in combination with the read/write (RW) sets, can be used for conflict resolution (§6). The challenge we tackle in this section is determining the boundaries of transactions (i.e., when a transaction's execution starts and finishes), which emerges from the execution of arbitrary function call-graphs §5.3.

5.1 Transaction Sequencing

In this section, we discuss the sequencing mechanism (❶) of Styx. Deterministic databases ensure the serializable execution of transactions by forming a global sequence. In Calvin [61], the authors propose a partitioned sequencer that retrieves the global sequence by communicating across all partitions, performing a deterministic round-robin.

Eliminating Sequencer Synchronization. Instead of the original sequencer of Calvin that sends $O(n^2)$ messages for the deterministic round-robin, Styx adopts a method similar to the one followed by Mencius [43], allowing Styx to acquire a global sequence without any communication between the sequencers ($O(1)$). This is achieved by having each sequencer assign unique transaction identifiers (TIDs) as follows:

$$TID_{sid,lc} = sid + (lc * n\_seq) \tag{1}$$

where $sid \in \mathbb{N}_1$ is the sequencer id assigned by the Styx coordinator in the registration phase, $lc \in \mathbb{N}_0$ is a local counter of each sequencer specifying how many TIDs it has assigned thus far and $n\_seq \in \mathbb{N}_1$ is the total number of sequencers in the Styx cluster. In the example of Figure 4, the sequencers of the three workers will sequence R_1, R_2, and R_3 to T_1, T_3, and T_2 respectively. Figure 5 illustrates how those TIDs are generated in parallel. Note that, conceptually, Styx implements a partitioned sequencer where the global sequence $S = \{S_1 \cup S_2 \cup \cdots \cup S_n\}$ is the union of all partitioned sequences.

Mitigating Sequence Imbalance. In case a single sequencer S_1 receives more traffic than other sequencers, its local counter (lc_1) will increase more than the local counter of the rest of the sequencers. As a result, in the next epoch, sequencer S_1 would produce larger TIDs than the rest of the sequencers. This means that new transactions arriving at a less busy sequencer will receive

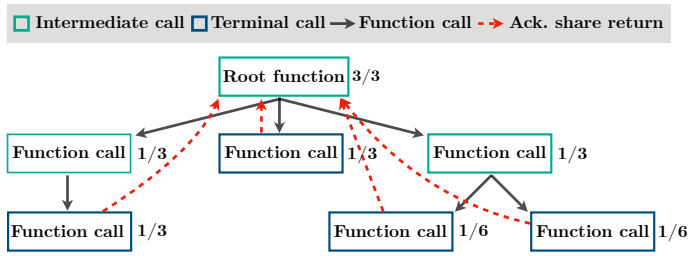


Fig. 6. Asynchronous function call chains. A given root function call may invoke other functions throughout its execution. The original acknowledgment (3/3) splits into parts as the function execution proceeds, and each function receives its own ack-share. For instance, in this function execution, the root function calls three other functions, thus splitting the ack-share into three equal parts. The same applies to subsequent calls, where the caller functions further split their ack-share. The sum of ack-shares of terminal (dark blue) calls (i.e., function calls that do not perform further calls) adds to exactly 3/3, which allows the root function to report the completion of execution.

higher priority for execution: transactions with higher TID receive less priority in our transactional protocol. In case of high contention in the workload, this would increase latencies for the busy (S_1) worker node. To avoid this, at the end of an epoch, the coordinator calculates the maximum lc ($max(lc_1, lc_2, \ldots, lc_n)$) and communicates it to all workers so that they can adjust their local counter re-balancing sequences in every epoch. Balancing the workers' transaction priorities reduces the 99th percentile latency.

Replication and Logging. There is no need to replicate and log the sequence within Styx since the input is logged and replicated within the replayable queue. In case of failure, after transaction replay, the sequencers will produce the exact same sequence (§7.2).

5.2 Call-Graph Discovery

After sequencing, Styx needs to execute the sequenced transactions and determine their call-graphs and RW sets (2). To this end, the function execution runtime ingests a given sequence of transactions to process in a given epoch. The number of transactions per epoch is either set by a polling interval or by a configurable maximum number of transactions that can run per epoch (by default, 1000 transactions per epoch). We have chosen an epoch-based approach since processing the incoming transactions in batches increases throughput.

Styx's runtime executes all the sequenced transactions on a snapshot of the data to discover the read/write sets. Transactions that span multiple workers will implicitly change the read/write sets of the remote workers via function calls. There is an additional issue related to discovering the RW set of a transaction: before the functions execute, the call-graph of the transaction is unknown. This is an issue because the protocol requires all transactions to be completed before proceeding to the next phase. To tackle this problem, Styx proposes a function acknowledgment scheme, which is explained in more detail in §5.3.

After this phase, all the stateful functions that comprise transactions will have finished execution, and the RW sets will be known. In Figure 4, transactions T_1, T_2, and T_3 will execute and create the following RW sets: $Worker_1 \rightarrow \{k_1 : T_1\}$, $Worker_2 \rightarrow \{k_2 : T_1, T_2 \text{ and } k_8 : T_2, T_3\}$ and $Worker_3 \rightarrow \{k_3 : T_3\}$.

5.3 Function Execution Acknowledgment

In the SFaaS paradigm, the call-graph formed by a transaction is unknown; functions could be coded by different developer teams and can form complex call-graphs. This uncertainty complicates determining when a transaction has completed processing, which is essential because phase 3 can only start after all transactions have finished processing. To that end, each asynchronous function call of a given transaction is assigned an `ack_share`. A given function knows how many shares to create by counting the number of asynchronous function calls during its runtime. The caller function then sends the respective acknowledgment shares to the downstream functions. For instance, in Figure 6, the transaction entry-point (root of the tree) calls three remote functions, splitting the ack_share into three parts (3 x 1/3). The left-most function invokes only one other function and passes to it its complete ack_share (1/3). The middle function does not call any functions, so it returns the share to the root function when it completes execution, and the right-most function calls two other functions, splitting its share (1/3) to 2 x 1/6. After all the function calls are complete, the root function should have collected all the shares. When the sum of the received shares adds to 1, the root/entry-point function can safely deduce that the execution of the entire transaction is complete.

This design is devised for two reasons: i) if every participating function just sent an ack when it is done, the root would not know how many acks to expect in order to decide whether the entire execution has finished, and ii) if we used floats instead of fractions we could stumble upon a challenge related to adding floating point numbers. For instance, if we consider floating-point numbers in the example given in Figure 6 that consists of the three function calls, the sum of all shares would not equal 1, but 0.99 since each share contributes 0.33. Subsequently, we cannot accurately round inexact division numbers; therefore, Styx uses fraction mathematics instead.

A solution close to the `ack_share` is the one of distributed futures [62]. However, it would not work in the SFaaS context as it either requires information about the entire call-graph for it to work asynchronously. Otherwise, it would need to create a chain of futures that would make it synchronous, introducing high latency.

6 Committing Transactions

After completing an epoch's call-graph discovery, Styx needs to determine which transactions will commit and which will abort based on the transactions' Read/Write (RW) sets and TIDs. To this end, this section presents two different commit phases: *i*) an optimistic lock-free phase that commits only the non-conflicting transactions, and *ii*) a lock-based phase that only commits the transactions that were not able to commit in the first phase. The lock-based commit phase commits all conflicting transactions by acquiring locks in a TID-ordered sequence. To make the second phase faster, we have devised a caching scheme to reuse the already-discovered call-graph to avoid re-executing long function chains whenever possible (§6.3).

6.1 Lock-free Commit Phase

In case of conflict (i.e., a transaction t writes a key that another transaction t' also reads or writes on), similarly to [42], only the transaction with the lowest transaction ID will succeed to commit (3). The transactions that have not been committed are put in a queue to be executed in the next phase 4 (maintaining their previously assigned ID).

In addition, workers (W) send their local conflicts to every other worker through the coordinator ($2 * |W|$ messages): this way, every worker retains a global view of all the aborted/rescheduled transactions and can decide, locally, which transactions can be committed. Finally, note that transactions can also abort, not because of conflicts but due to application logic failures (e.g., by

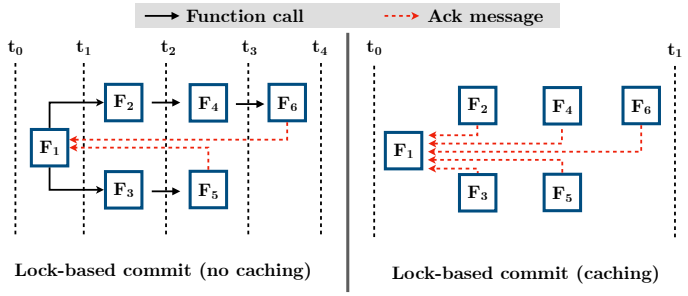


Fig. 7. If no function caching is performed (left), the transaction execution will execute a deep call-graph; the messages will be sent sequentially and be equal to the number of function calls (5) in addition to the acks (2). Styx's function caching optimization (right) will lead to a concurrent function execution in the lock-based commit phase, between t_0 and t_1, and send only five acks asynchronously.

throwing an exception due to an integrity constraint violation). In that case, Styx removes the related entries from the read/write sets to reduce possible conflicts.

In this phase, all the transactions that have not been part of a conflict apply their writes to the state, commit, and reply to the clients. In the example shown in Figure 4, only T_1 can commit in W_1 and W_2 due to conflicts in the RW sets of W_2 regarding T_2 and T_3; more specifically, at keys k_2 and k_8.

6.2 Lock-based Commit Phase

In the previous phase, (3), only transactions without conflicts can be committed. We now explain how Styx deals with transactions that have not been committed in a given epoch due to conflicts ((4)). First, Styx acquires locks in a given sequence ordered by transaction ID. Then, it reruns all transactions concurrently since all the read/write sets are known and commits them. However, if a transaction's read/write set changes in this phase, Styx aborts the transaction and recomputes its read/write set in the next epoch. Now, in Figure 4, W_2 can sequentially acquire locks for T_2 and T_3, leading to their commits in W_2 and W_3.

6.3 Call-Graph Caching

As depicted in Figure 4, the lock-based commit phase (4) is used to execute any transactions that did not commit during the lock-free commit phase (3). By the time the lock-based commit phase starts, the state of the database may have changed since the lock-free commit. As a result, function invocations need to be re-executed to account for the data updates.

On the left part of Figure 7, we depict such a function invocation. At time t_0, F_1 is invoked, which in turn invokes two function chains: $F_1 \rightarrow F_2 \rightarrow F_4 \rightarrow F_6$ and $F_1 \rightarrow F_3 \rightarrow F_5$. Once the two function chains finish their execution (on time t_4 and t_3 respectively), they can acknowledge their termination to the root call F_1.

Potential for Caching. During our early experiments, we noticed cases where F_1 is invoked and the parameters with which it calls F_2 (and in turn the invocations across the $F_1 \rightarrow ... \rightarrow F_6$ call chain) do not change. The same applied to the RW set of those function invocations; the RW sets remained unchanged. Since Styx tracks those call parameters as well as the functions' RW sets, it can cache input parameters during the lock-free commit phase and reuse them during the lock-based commit, avoiding long sequential re-executions along the call chains. This case is depicted on the right part of Figure 7: the function-call chain does not need to be invoked in a

sequential manner from F_1 all the way to F_6, leading to high latency. Instead, the individual workers can re-invoke those function calls locally and concurrently. As a result, all functions can execute in parallel and save on latency and network overhead ($t_4 - t_1$ in Figure 7). Furthermore, caching does not require user input, is transparent to the API, and does not depend on the synchronous or asynchronous specification. Nonetheless, synchronous calls can be automatically transformed into asynchronous ones under certain conditions [4, 50].

Conditions for Parallel Function Re-invocation. Intuitively, if the parameters with which, e.g., F_2 is called, and the RW set of F_2 remains the same, we can safely assume that function F_2 can be invoked concurrently without having to be invoked sequentially by F_1. If those functions are successfully completed and acknowledge their completion to the root function F_1, it means that the transaction can be committed. To the contrary, if the RW set of any of the functions $F_1 - F_6$ changes, or the parameters of any of the functions along the call chains change, the transaction must be fully re-executed. In that case, Styx will have to reschedule that transaction to the next epoch.

6.4 Early Commit-Replies via Determinism

Implementing Styx as a fully deterministic dataflow system offers a set of advantages involving the ability to communicate transaction commits to external systems (e.g., the client) even before the state snapshots are persisted to durable storage. A traditional transactional system can respond to the 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. In Styx's case, that would mean when an asynchronous snapshot completes (i.e., is persisted to durable storage such as S3), leading to high latency.

Since Styx implements a deterministic transactional protocol executing 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 enables Styx to give early commit-replies: *the client can receive the reply even before a persistent snapshot is stored.* The assumption here is that the input queue, persisting the client requests, will provide to Styx's sequencers the requests in the same order after replay, a guarantee that is typically provided by most modern message brokers. Performing state mutations and message passing before persistence has also been explored in DARQ's speculative execution [38].

7 Fault Tolerance

Styx implements a coarse-grained fault tolerance mechanism. Instead of logging each function execution, it adopts a variant of existing checkpointing mechanisms used in streaming dataflow systems [7, 9, 53]. Styx asynchronously snapshots state and stores it in a replicated fault-tolerant blob store (e.g., Minio / S3), enabling low-latency function execution. We describe Styx's fault tolerance mechanism below.

7.1 Incremental Snapshots & Recovery

The snapshotting mechanism of Styx resembles the approach of many streaming systems [3, 7, 21, 27] that extend the seminal Chandy-Lamport snapshots [9]. Modern stream processing systems checkpoint their state by receiving snapshot barriers at regular time intervals (epochs) decided by the coordinator. In contrast, Styx leverages an important observation: Workers do not need to wait for a barrier to enter the system in order to take a snapshot since the natural barrier in a transactional epoch-based system like Styx is at the end of a transaction epoch.

Algorithm 1: Snapshotting Mechanism

Result: Compacted Snapshot stored in durable storage
Input : δ: Delta changes, O_{input}: Input offset, O_{output}: Output offset, E_{count}: Epoch count, SEQ_{count}: Sequence count
Output : $\mathcal{S}$: Compacted snapshot

```
if snapshotInterval then
    state ← δ                                          ▷ Prepare data and metadata for snapshot
    metadata ← {O_input, O_output, E_count, SEQ_count}
    S^δ ← serialize(state, metadata)
    store S^δ
    inform coordinator
end
if compactionInterval then
    S ← ∅
    foreach S^δ do
        S ← compact(S, S^δ)                            ▷ Compact delta snapshots
    end
end
```

Algorithm 2: Recovery Mechanism

Result: Recovered state from durable storage, possible duplicate messages
Input : $\mathcal{S}$: Latest compacted snapshot,
$\mathcal{S}^{\delta}$: Incremental (delta) snapshots,
O^{last}_{output}: Offset of last output,
Output : $\mathcal{R}$: Set of possible duplicate messages, $state^s$: Snapshotted state, O^s_{input}: Snapshotted input offset, O^s_{output}: Snapshotted output offset, E^s_{count}: Snapshotted epoch count, SEQ^s_{count}: Snapshotted sequencer count

```
if S^δ ≠ ∅ then
    S ← compact(S, S^δ)                                ▷ Compact delta snapshots, if any
end
state^s, O^s_input, O^s_output, E^s_count, SEQ^s_count ← deserialize S'
                                                       ▷ Extract persisted state
R ← {m | O^s_output ≤ m ≤ O^last_output}                ▷ Possible duplicates (§7.4)
```

Snapshotting. To this end, instead of taking snapshots periodically by propagating markers across the system's operators, Styx aligns snapshots with the completion of transaction epochs to take a consistent cut of the system's distributed state, including the state of the latest committed transactions, the offsets of the message broker, and the sequencer counters (lc). The minimal information included in the snapshot is $O(N + c)$, where N is the number of updates affecting the delta map, and c is the fixed number of integers stored for the Kafka offsets and the sequencer variables.

When the snapshot interval triggers, Styx makes a copy of the current state changes to a parallel thread and persists incremental snapshots asynchronously, allowing Styx to continue processing incoming transactions while the snapshot operation is performed in the background. The snapshotting procedure is described in Algorithm 1.

Recovery. In case of a system failure, Styx *i*) rolls back to the epoch of the latest completed snapshot, *ii*) loads the snapshotted state, *iii*) rolls back the replayable source's topic partitions (that are aligned with the Styx operator partitions) to the offsets at the time of the snapshot, *iv*) loads

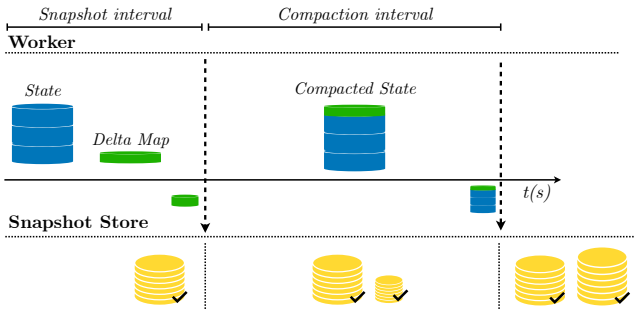


Fig. 8. Incremental snapshots with Delta Maps in Styx.

the sequencer counters, and finally, v) verifies that the cluster is healthy before executing a new epoch. The recovery procedure is described in Algorithm 2.

Incremental Snapshots & Compaction. Each snapshot stores a collection of state changes in the form of *delta maps*. A delta map is a hash table that tracks the changes in a worker's state in a given snapshot interval. When a snapshot is taken, only the delta map containing the state changes of the current interval is snapshotted. To avoid tracking changes across delta maps, Styx periodically performs compactions where the deltas are merged in the background, as shown in Figure 8. The cost of compacting is equivalent to the cost of merging two hashmaps with the same key-spaces ($O(N)$). The total cost will be $O(M * N)$, with M denoting the number of deltamaps we need to compact.

7.2 Sequencer Recovery

To guarantee determinism, upon recovery, Styx's sequencer needs to generate identical sequences as the ones generated between the latest snapshot and failure. The recovery protocol of the sequencer operates as follows: At first, during the snapshot, we store the local counter of each sequencer partition (lc) with its id (sid) and the epoch counter. Additionally, at the start of each epoch, Styx logs the number of transactions contained in that epoch, denoted as epoch size. Logging the epoch sizes is needed due to Styx's varying epoch sizes and the sequencer rebalancing scheme (§5.1). After failure, the recovered sequencer partitions are initialized with the snapshot's lc and sid. Afterward, each partition retrieves from its log all the sizes of all epochs executed since the last snapshot. Finally, after recovery, the sequencer matches the epoch sizes to the ones recorded by the log, leading to the same global sequence observed before failure.

7.3 Exactly-Once Processing

At first, the durable input queue, which acts as a replayable source, allows Styx to replay requests after failures. By rolling back the queue partitions (aligned with system operator partitions) to the respective offsets as recorded in the latest snapshot, Styx can reprocess only the transactions whose state changes are not reflected yet in the snapshot. Transactions committed and early commit-replies stored in the egress can be deduplicated (§7.4).

Styx runs each transaction to its completion in a single epoch. A given transaction can execute a large call-graph of functions that can affect the state. If a failure occurs, a transaction's state effects are restored to the latest snapshot, and the complete transaction is re-executed. As a result, no special attention is required to ensure that remote function calls are executed exactly-once, except for resetting all TCP channels between Styx's workers after recovery.

Lemma 7.1. *The state mutations of committed transactions in Styx are reflected exactly-once, even upon failure.*

Proof. Let S_t denote the state of the system at time t. $Q_t = \{r_1, \ldots, r_n\}$ denotes the durable input queue at time t that holds all requests r_i to be processed. We assume that the input queue operates as FIFO and requests r_i are deterministic. Each r_i will be sequenced as a transaction $T_i = \{upd_l, func_m\}$ by a deterministic sequencer, where upd_l are the state updates and $func_m$ are the function calls of the transaction. We assume that upd_l happens atomically and $func_m$ is also reflected once, given the use of a reliable communication protocol. Given the same initial state S and input from Q, it always produces the same state transition $S \rightarrow S'$,which means $S'_{t+1} = mutation(S_t, Q_t)$. The execution of a transaction T_i is deterministic.

At any time t, the state of the system S_t reflects all transactions in Q_t that have been fully executed and committed. Accordingly, the state S_t ignores partially executed or in-progress transactions in Q_t. We denote the latest durable snapshot taken up to time t, as Snapshot(S_t, i, n) where n corresponds to the offsets of the first request r_i, and last request r_n of the input queue to be processed up to time t. Upon failure, a subset of Q_t, $Q_t^{success} = \{r_1, \ldots, r_k\}$ will contain successfully committed transactions and a subset $Q_t^{fail} = \{r_{k+1}, \ldots, r_n\}$ will contain aborted transactions such that $Q_t = Q_t^{success} + Q_t^{fail}$. In order to recover from a failure, Q_t is rolled back to S_t from Snapshot(S_t, i, n) as we persist the offsets of our input queue. Transactions in Q_t are replayed in the original order from offset i to offset n of our input queue. This is ensured by the FIFO queue and the deterministic sequencer. After processing the input transactions, $Q_t^{success}$ includes requests already reflected in Snapshot(S_t), and Q_t^{fail} includes pending requests. Since Snapshot(S_t) reflects $Q_t^{success}$ and $Q_t = Q_t^{success} + Q_t^{fail}$, the replay and processing ensure: $S''_{t+1} = mutation(S_t, Q_t^{fail}) = S'_{t+1}$. Thus, the effects of all transactions will be reflected in the state exactly-once, even after failure. □

7.4 Exactly-Once Output

A common challenge in the fault tolerance of streaming systems is that of the exactly once output [15, 19] in the presence of failures, which is hard to solve for low-latency use cases. For example, in Apache Flink's [8] exactly-once output configuration, clients can only retrieve responses after those are persisted in a snapshot or a transactional sink. This arrangement is sufficient for streaming analytics but not for low-latency transactional workloads, as discussed previously in §6.4.

To solve that, during recovery, Styx: *i*) reads the last offset of the egress topic, *ii*) compares it with the output offset persisted in the snapshot, determining for which transactions the clients have already received replies, *iii*) retrieves the TIDs attached in those replies, and *iv*) does not send a reply again to the egress topic for those transactions. Note that this deduplication strategy is based on the fact that TIDs have been assigned deterministically.

7.5 Addressing Non-Deterministic Functions

As discussed in §7.1 Styx's recovery mechanism is based on deterministic replay. To this end, Styx requires that the functions authored by developers are also deterministic, i.e., replaying the same function multiple times, using the same inputs and database state, should yield the same results. However, one can achieve determinism even in the presence of non-deterministic logic inside functions, such as randomness (e.g., random numbers/sampling) or calls to external systems (e.g., calling an external database or API). Styx can follow the approach of existing systems (e.g., Temporal [60], Clonos [53]). In the following, we explain how this can be achieved.

Randomness. To retain determinism in the case of randomness, Styx can use an external fault tolerant write-ahead log (WAL) to log the random number along with the TID. Thus, in the case of

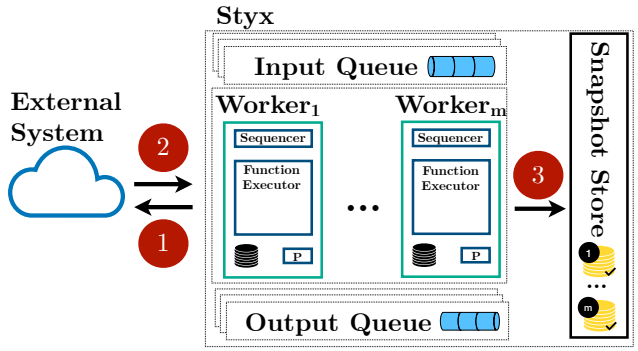


Fig. 9. External system call critical points and Styx.

failure and replay, Styx can use the logged random number, essentially making the function call deterministic during replay.

Calls to External Systems. As illustrated in Figure 9, an interaction with an external system needs to consider three critical points to maintain determinism. Styx assumes that the external system supports idempotency [26], meaning that if a call is made twice with the same idempotency key, the effects on the external system's state and its return value will remain the same. In (1) Styx needs to log the idempotency key and the TID in the WAL before calling the external system. If the external system produces a response ((2)), Styx can store it in the WAL and retrieve it from there in case of replay. Finally, when Styx completes a snapshot ((3)), it can also clear the WAL for garbage collection since the prior entries are not needed.

Finally, Styx could mask those operations behind an API that exposes the following functionality, such as `styx.random` for random number generation and `styx.call_external` for external system calls.

8 Evaluation

We evaluate Styx by answering the following questions:

- (§8.2) How does Styx compare to State-of-the-Art serializable transactional SFaaS systems?
- (§8.2) How does Styx perform under skewed workload?
- (§8.3) How well does Styx scale?
- (§8.4) Does the snapshotting mechanism affect performance?

8.1 Setup

Systems Under Test. In the evaluation, we include SFaaS systems that provide serializable transactional guarantees. Those are:
Beldi [63]/Boki [28]. Both systems use a variant of two-phase commit and Nightcore [29] as their function runtime and store their data in DynamoDB. Additionally, Boki is deployed with the latest improvements of Halfmoon [51].
T-Statefun [12]. T-Statefun maintains the state and the coordination of the two-phase commit protocol within an Apache Flink cluster and ships the relevant state to remote stateless functions for execution. For fault tolerance, it relies on a RocksDB state backend that performs incremental snapshots.

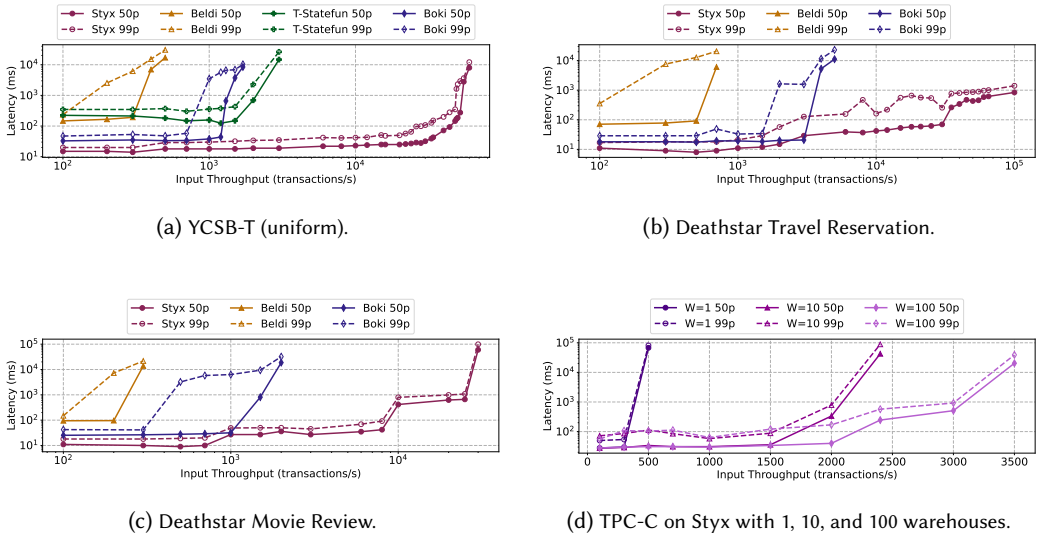
Scenario	#keys	Function Calls	Transactions %
YCSB-T	10k	2	100%
Deathstar Movie	2k	9-10	0%
Deathstar Travel	2k	3	0.5%
TPC-C	1m-100m	8 / 20-50	100%

Table 1. Workload characteristics.

Styx. Styx is implemented in Python 3.12 and uses coroutines to enable asynchronous concurrent execution. Apache Kafka is used as an ingress/egress, and Minio/S3 [45] is used as a remote persistent store for Styx's incremental snapshots. Finally, Styx is a standalone containerized system that works on top of Docker and Kubernetes for ease of deployment.

Workloads/Benchmarks. Table 1 summarizes the three workloads used in the experiments.
YSCB-T [14]. We use a variant of YCSB-T [14] where each transaction consists of two reads and two writes. The concrete scenario is as follows: First, we create 10.000 bank accounts (keys) and perform transactions in which a debtor attempts to transfer credit to a creditor. This transfer is subject to a check on whether the debtor has sufficient credit to fulfill the payment. If not, a rollback needs to be performed. The selection of a relatively small number of keys is deliberate: we want to assess the systems' ability to sustain transactions under high contention. In addition, for the experiment depicted in Figure 11 (skewed distribution), we select the debtor key based on a uniform distribution and the creditor based on a Zipfian distribution, where we can vary the level of contention by modifying the Zipfian coefficient.
Deathstar [20]. We employ Deathstar [20], as adapted to SFaaS workloads by the authors of Beldi [63]. It consists of two workloads: *i*) the Movie workload implements a movie review service where users write reviews about movies, *ii*) the Travel workload implements a travel reservation service where users search for hotels and flights, sort them by price/distance/rate, find recommendations, and transactionally reserve hotel rooms and flights. Both Deathstar workloads follow a uniform distribution. Note that T-Statefun could not run in this set of experiments since it does not support range queries.
TPC-C [37]. The prime transactional benchmark targeting OLTP systems is TPC-C [37]. In our evaluation, we employ the NewOrder and Payment transactions, and we had to rewrite them into the SFaaS paradigm, 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 the contention, allowing for higher throughput and lower latency. Note that the TPC-C experiments do not include Beldi, Boki, or T-Statefun because they do not support it.

Resources. For Beldi/Boki, T-Statefun and Styx, we assigned a total of 112 CPUs with 2GBs of RAM per CPU, matching what is presented in the original Boki paper [28]. Additionally, throughout all the evaluation scenarios, the data fit in memory across all systems. Unless stated otherwise, Styx and T-Statefun are configured to perform incremental snapshots every 10 seconds. All external systems, i.e., DynamoDB (Beldi, Boki), Minio, and Kafka (Styx, T-Statefun), are configured with three replicas for fault tolerance.



(a) YCSB-T (uniform).

(b) Deathstar Travel Reservation.

(c) Deathstar Movie Review.

(d) TPC-C on Styx with 1, 10, and 100 warehouses.

Fig. 10. Evaluation in different scenarios. T-Statefun does not support range queries required by the Deathstar workloads. TPC-C is only supported by Styx.

External Systems. Boki and Beldi use a fully managed DynamoDB instance at AWS, which does not state the amount of resources it occupies and is additional to the 112 CPUs assigned to Boki and Beldi. Similarly, the resources assigned to Minio/S3 (Styx and T-Statefun) are not accounted for.

Metrics. Our goal is to observe systems' behavior, measured by their latency while varying the input throughput.
Input throughput represents the number of transactions submitted per second to the system under test. As the input throughput increases during an experiment, we expect the latency of individual transactions to increase until aborts start to manifest due to contention or high load.
Latency represents the time interval between submitting a transaction and the reported time when the transaction is committed/aborted. In Styx and T-Statefun, the latency timer starts when a transaction is submitted in the input queue (Kafka) and stops when the system reports the transaction as committed/aborted in the output queue. Similarly, in Beldi and Boki, the latency is the time since the input gateway has received a transaction and the time that the gateway reports that the transaction has been committed/aborted.

8.2 Latency vs. Throughput

We first study the latency-throughput tradeoff of all systems. We retain the resources given to the systems constant (112 CPUs) while progressively increasing the input throughput. We measure the transaction latency. As depicted in Figure 10, Styx outperforms its baseline systems by at least an order of magnitude. Specifically, in YCSB-T (Figure 10a), Styx achieves a performance improvement of ~20x in terms of throughput against T-Statefun, which ranks second. In addition, Styx outperforms Boki by ~30x in Deathstar's travel reservation workload (Figure 10b) and by ~35x in Deathstar's movie review Figure 10c) workload. Finally, in the TPC-C benchmark (Figure 10d), which requires a large number of function calls per transaction (20-50), we observe that Styx's

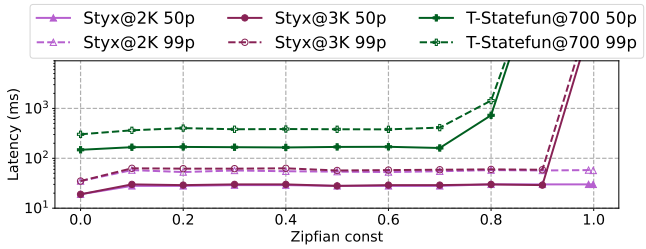


Fig. 11. Latency evaluation for varying levels of contention (0.0 - 0.999) with YCSB-T (skewed). We ran Styx with two different input throughput variations to show its behavior under contention clearly. Note that Styx and T-Statefun execute all transactions to completion (abort%=0).

		0.0	0.2	0.4	0.6	0.8	0.9	0.99	0.999
Beldi	Abort %	47.93	45.54	44.31	47.28	52.40	56.06	61.62	60.70
	CMT TPS	104	108	111	105	95	76	76	78
Boki	Abort %	48.77	48.23	49.54	51.82	61.29	68.50	74.47	70.71
	CMT TPS	359	362	353	337	271	220	179	205

Table 2. Evaluation of Boki and Beldi for varying levels of contention with YCSB-T. We report the abort ratio and committed transactions rate and omit latency since the systems do not execute all transactions to completion. Both run at their maximum sustainable throughput.

performance improves as we increase the input throughput for different numbers of warehouses, reaching up to 3K TPS with sub-second 99^{th} percentile latency (100 warehouses).

Aborts & Throughput. Beldi and Boki follow a no-wait-die concurrency control approach, which leads to a significant amount of aborts as the throughput increases. Styx and T-Statefun do not use such a transaction abort mechanism. Instead, they execute all transactions to completion. This difference in handling transactions under high load makes the latencies across systems hard to compare. For this reason, in Figure 11, we plot the results of Styx and T-Statefun and present the performance of Beldi and Boki in a separate table (Table 2), alongside their abort rates.

We observe the following: *i*) at the highest level of contention ($Zipfian$ at 0.999) Styx achieves at least 2000 TPS, outperforming the rest by ~5-10x in terms of effective throughput, *ii*) both Beldi and Boki (that run at their maximum sustainable throughput) abort more transactions as the level of contention increases (~40-70%), which significantly impacts their effectiveness as shown in Table 2, and *iii*) Styx shows an increase in latency only in high levels of contention ($Zipfian > 0.99$) while executing at ~4x higher throughput than the rest.

Runtime Breakdown. In Table 3, we show where the systems under test spend their processing time. We use YCSB-T for this purpose since it is the only benchmark supported by all the systems (§8.1). We measured the median latency while all the systems were running at 100 TPS for 60 seconds and averaged the proportions of function execution, networking, and state access across all committed transactions. The key observations are: *i*) Styx's co-location of processing and state led to minimal state access latency, and *ii*) Styx's asynchronous networking allows for lower network latency.

Takeaway. The rather large performance advantages of Styx across all experiments are enabled by the following three properties and design choices: *i*) the co-location of processing and state with

System	Function Execution	Networking	State Access
Styx	**0.34ms** - 2.2%	**14.33ms** - 95.6%	**0.32ms** - 2.2%
Boki	1.1ms - 3.3%	16.1ms - 49%	15.68ms - 47.7%
T-Statefun	2.76ms - 2.2%	92.12ms - 74.3%	29.11ms - 23.5%
Beldi	1.01ms - 0.7%	56.58ms - 38.4%	89.57ms - 60.9%

Table 3. Performance breakdown of all systems. (median latency - percentage from the total)

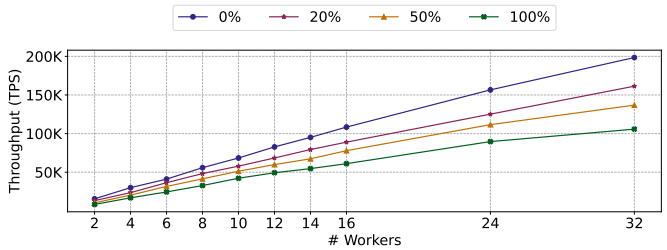


Fig. 12. Scalability of Styx on YCSB-T with varying percentages of multi-partition transactions.

efficient networking as shown in Table 3, contrary to the other systems that have to transfer the state to their function execution engines; *ii*) the asynchronous snapshots with delta maps for fault tolerance compared to the replication of Beldi/Boki and the LSM-tree-based incremental snapshots of T-Statefun; *iii*) the efficient transaction execution protocol employed in Styx compared to the two-phase commit used by Styx's competition.

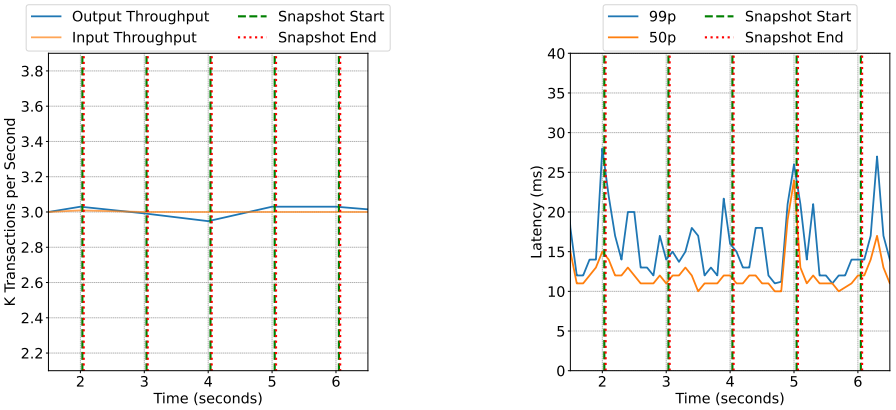
8.3 Scalability

In this experiment, we test the scalability of Styx by increasing the number of Styx workers. Each worker is assigned 1 CPU and a state of 1 million keys. We measure the maximum throughput on YCSB-T. 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 12, we observe that in all settings, Styx retains near-linear scalability. Finally, Styx displays the expected behavior as the number of multi-partition transactions increases.

8.4 Fault-Tolerance Evaluation

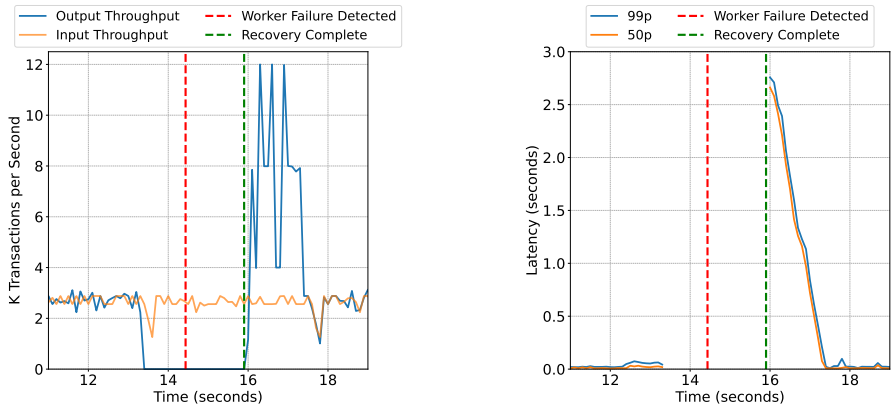
Effect of Snapshots. In Figure 13, we depict the impact of the asynchronous incremental snapshots to Styx's performance. In both figures, we mark when a snapshot starts and ends. The state includes 1 million keys, and we use a 1-second snapshot interval. Styx is deployed with four 1-CPU workers, and the input transaction arrival rate is fixed to 3K YCSB-T TPS. In Figure 13a, we observe that during a snapshot operation, Styx shows virtually no performance degradation in throughput. In Figure 13b, we observe a minor increase in the end-to-end latency in some snapshots. The reason for that is the concurrent snapshotting thread, which competes with the transaction execution thread during snapshotting. At the same time, it also has to block the transaction execution thread momentarily to copy the corresponding operator's state delta.

Recovery Time. In Figure 14, we evaluate the recovery process of Styx with the same parameters as in Figure 13. We reboot a Styx worker at ~13.5 seconds. It takes Styx's coordinator roughly a second



(a) Throughput (b) Latency

Fig. 13. Impact of Styx's snapshotting on performance



(a) Throughput (b) Latency

Fig. 14. Styx's behavior during recovery.

to detect the failure. Then, after the reboot, the coordinator re-registers the worker and notifies all workers to load the last complete snapshot, merge any uncompacted deltas, and use the message broker offsets of that snapshot. The recovery time is also observed in the latency (Figure 14b) that is ~2.5 seconds (time to detect the failure in addition to the time to complete recovery). In terms of throughput (Figure 14a), we observe Styx working on its maximum throughput after recovery completes to keep up with the backlog and the input throughput.

Effect of Large State Snapshots. In Figure 15, we test the incremental snapshotting mechanism against a larger state of 20 GB from TPC-C using a bigger Styx deployment of 100 1-CPU workers at 10-second checkpoint intervals. From 0 to the 750-second mark, Styx is importing the dataset. Since there are no small deltas (importing is an append-only operation), snapshotting is more expensive than the normal workload execution, where only the deltas are stored in the snapshots. The increase in latency at ~550 seconds corresponds to the loading of the largest tables (Stock and

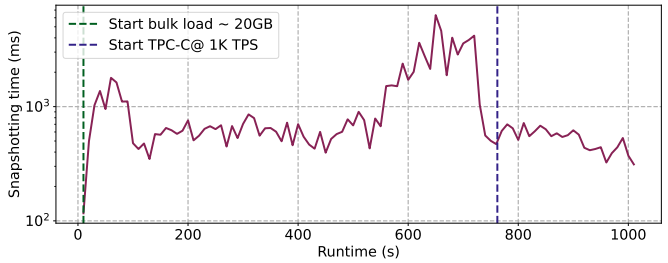


Fig. 15. Behaviour of incremental snapshots on Styx with ~20GB TPC-C state.

Order-Line) in the system. After loading the data and starting the transactional workload at 1000 TPS, we observe a drop in latency due to fewer state changes within the delta maps.

9 Related Work

Transactional SFaaS. SFaaS has received considerable research attention and open-source work. Transactional support with fault tolerance guarantees (that popularized DBMS systems) is necessary to widen the adoption of SFaaS. Existing systems fall into two categories: i) those that focus on transactional serializability and ii) those that provide eventual consistency. The first category includes Beldi [63], Boki [28], and T-Statefun [12]. Beldi implements linked distributed atomic affinity logging on DynamoDB to guarantee serializable transactions among AWS Lambda functions with a variant of the two-phase commit protocol. Boki extends Beldi by adding transaction pipeline improvements regarding the locking mechanism and workflow re-execution. In turn, Halfmoon [51] extends Boki with an optimal logging implementation. T-Statefun [12] also uses two-phase commit with coordinator functions to support serializability on top of Apache Flink's Statefun. For eventually-consistent transactions, T-Statefun implements the Saga pattern. Cloudburst [56] also provides causal consistency guarantees within a DAG workflow. Proposed more recently, Netherite [5] offers exactly-once guarantees and a high-level programming model for Microsoft's Durable Functions [6], but it does not guarantee transactional serializability across functions. Unum [41] needs to be paired with Beldi or Boki to ensure end-to-end exactly-once and transactional guarantees.

Dataflow Systems. Support for fault-tolerant execution in the cloud with exactly-once guarantees [7, 16] is one of the main drivers behind the wide adoption of modern dataflow systems. However, they lack a general and developer-friendly programming model with support for transactions and a natural way to program function-to-function calls. Closer to the spirit of Styx are Ciel [47] and Noria [22]. Ciel proposes a language and runtime for distributed fault-tolerant computations that can execute control flow. Noria solves the view maintenance problem via a dataflow architecture that can propagate updates to clients quickly, targeting web-based, read-heavy computations. However, neither of the two provides a transactional model for workflows of functions like Styx.

Transactional Protocols. Besides Aria [42], which inspired the protocol we created for Styx (§4), two other protocols fit the requirement of no a priori read/write set knowledge: Starry [65] and Lotus [66].. Starry targets replicated databases with a semi-leader protocol for multi-master transaction processing. At the same time, Lotus [66] focuses on improving the performance of multi-partition workloads using a new methodology called run-to-completion-single-thread (RCST).

Styx makes orthogonal contributions to these works and could adopt multiple ideas from them in the future.

10 Future Work

Elasticity in Dataflow Systems. Extensive work has been carried out in dynamic reconfiguration [17, 22, 30] and state migration [13, 24, 25] of streaming dataflow systems over the last few years. These advancements are necessary for providing serverless elasticity in the case of state and compute collocation to leverage dataflows as an execution model for serverless stateful cloud applications, which is a future goal of Styx.

Replication for High Availability. In the Styx architecture, replication is only applied in the snapshot store and the Input/Output queues to ensure fault tolerance. For high-availability, Styx could adopt replication mechanisms from deterministic databases. Specifically, the design of deterministic transaction protocols, such as Calvin [61], feature state replicas that require no explicit synchronization. First, the sequencer replicas need to agree on the order of execution. After that, the deterministic sequencing algorithm guarantees that the resulting state will be the same across partition/worker replicas by all replicas executing state updates in the same order.

Non-Deterministic Functions on Streaming Dataflows. In its current version, Styx requires application logic to be deterministic, similar to OLTP [31, 58], where stored procedures are required to be deterministic since they run independently on different replicas. The same determinism requirement applies to SFaaS [12, 28] systems. However, real-world applications may encapsulate logic that makes the outcome of their execution non-deterministic. Examples of non-deterministic operations are calls to external systems and using random number generators or time-related activities. That said, we have a plan for supporting non-deterministic functions in Styx, as discussed in §7.5.

11 Conclusion

This paper presented Styx, a distributed streaming dataflow system that supports multi-partition transactions with serializable isolation guarantees through a high-level, standard Python programming model that obviates transaction failure management, such as retries and rollbacks. Styx follows the deterministic database paradigm while implementing a streaming dataflow execution model with exactly-once processing guarantees. Styx outperforms the state-of-the-art by at least one order of magnitude in all tested workloads regarding throughput.

Acknowledgments

We want to thank Paris Carbone for his advice throughout the process of developing Styx and the anonymous reviewers for their constructive feedback. 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, 78–88. doi:10.1145/3181853

[2] Amazon. 2025. AWS Step Functions. https://aws.amazon.com/step-functions Accessed on April 07, 2025.

[3] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 601–613. doi:10.1145/3183713.3190664

[4] Sidi Mohamed Beillahi, Ahmed Bouajjani, Constantin Enea, and Shuvendu K. Lahiri. 2022. Automated Synthesis of Asynchronizations. In *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13790)*, Gagandeep Singh and Caterina Urban (Eds.). Springer, 135–159. doi:10.1007/978-3-031-22308-2_7

[5] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: Efficient Execution of Serverless Workflows. *Proc. VLDB Endow.* 15, 8 (2022), 1591–1604. doi:10.14778/3529337.3529344

[6] 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 (2021), 1–27. doi:10.1145/3485510

[7] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12, 1718–1729. doi:10.14778/3137765.3137777

[8] 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).

[9] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1, 63–75. doi:10.1145/214451.214456

[10] 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 Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p30-cheng.pdf

[11] 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

[12] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Distributed transactions on serverless stateful functions. In *15th ACM International Conference on Distributed and Event-based Systems, DEBS 2021, Virtual Event, Italy, June 28 - July 2, 2021*, Alessandro Margara, Emanuele Della Valle, Alexander Artikis, Nesime Tatbul, and Helge Parzyjegla (Eds.). ACM, 31–42. doi:10.1145/3465480.3466920

[13] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient management of very large distributed state for stream processing engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2471–2486.

[14] Akon Dey, Alan D. Fekete, Raghunath Nambiar, and Uwe Röhm. 2014. YCSB+T: Benchmarking web-scale transactional databases. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*. IEEE Computer Society, 223–230. doi:10.1109/ICDEW.2014.6818330

[15] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* 34, 3 (2002), 375–408. doi:10.1145/568522.568525

[16] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. 2014. Making State Explicit for Imperative Big Data Processing. In *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 49–60. https://www.usenix.org/conference/atc14/technical-sessions/presentation/castro-fernandez

[17] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-Regulating Stream Processing in Heron. *Proc. VLDB Endow.* 10, 12 (2017), 1825–1836. doi:10.14778/3137765.3137786

[18] The Apache Software Foundation. 2025. Apache Airflow. https://airflow.apache.org Accessed on April 07, 2025.

[19] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2024. A survey on the evolution of stream processing systems. *VLDB J.* 33, 2 (2024), 507–541. doi:10.1007/S00778-023-00819-8

[20] 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.

[21] Can Gencer, Marko Topolnik, Viliam Durina, Emin Demirci, Ensar B. Kahveci, Ali Gürbüz, József Bartók, Grzegorz Gierlach, Frantisek Hartman, Ufuk Yilmaz, Ondrej Lukás, Mehmet Dogan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Hazelcast Jet: Low-latency Stream Processing at the 99.99th Percentile. *Proc. VLDB Endow.* 14, 12, 3110–3121. doi:10.14778/3476311.3476387

[22] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10,*

2018, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 213–231. https://www.usenix.org/conference/osdi18/presentation/gjengset

[23] Jim Gray. 1978. Notes on Data Base Operating Systems. 60 (1978), 393–481. doi:10.1007/3-540-08755-9_9

[24] Rong Gu, Han Yin, Weichang Zhong, Chunfeng Yuan, and Yihua Huang. 2022. Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 539–556. https://www.usenix.org/conference/atc22/presentation/gu-rong

[25] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. 2019. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *Proc. VLDB Endow.* 12, 9 (2019), 1002–1015. doi:10.14778/3329772.3329777

[26] IETF. 2025. The Idempotency-Key HTTP Header Field. https://www.ietf.org/archive/id/draft-ietf-httpapi-idempotency-key-header-06.txt Accessed on April 07, 2025.

[27] Gabriela Jacques-Silva, Fang Zheng, Daniel Debrunner, Kun-Lung Wu, Victor Dogaru, Eric Johnson, Michael Spicer, and Ahmet Erdem Sariyüce. 2016. Consistent Regions: Guaranteed Tuple Processing in IBM Streams. *Proc. VLDB Endow.* 9, 13, 1341–1352. doi:10.14778/3007263.3007272

[28] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 691–707. doi:10.1145/3477132.3483541

[29] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 152–166.

[30] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava C. Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 783–798. https://www.usenix.org/conference/osdi18/presentation/kalavri

[31] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.* 1, 2, 1496–1499. doi:10.14778/1454159.1454211

[32] Tom Killalea. 2016. The hidden dividends of microservices. *Commun. ACM* 59, 8 (2016), 42–45. doi:10.1145/2948985

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

[34] Rodrigo N. 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 (2021), 3348–3361. doi:10.14778/3484224.3484232

[35] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 382–401. doi:10.1145/357172.357176

[36] Andrea Lattuada, Frank McSherry, and Zaheer Chothia. 2016. Faucet: a user-level, modular technique for flow control in dataflow engines. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2016, San Francisco, CA, USA, July 1, 2016*, Foto N. Afrati, Jacek Sroka, and Jan Hidders (Eds.). ACM, 2. doi:10.1145/2926534.2926544

[37] Scott T. Leutenegger and Daniel M. Dias. 1993. A Modeling Study of the TPC-C Benchmark. (1993), 22–31. doi:10.1145/170035.170042

[38] Tianyu Li, Badrish Chandramouli, Sebastian Burckhardt, and Samuel Madden. 2023. DARQ Matter Binds Everything: Performant and Composable Cloud Programming via Resilient Steps. *Proc. ACM Manag. Data* 1, 2 (2023), 117:1–117:27. doi:10.1145/3589262

[39] Tianyu Li, Badrish Chandramouli, Sebastian Burckhardt, and Samuel Madden. 2024. Serverless State Management Systems. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org. https://www.cidrdb.org/cidr2024/papers/p16-li.pdf

[40] Lightbend. 2025. Akka.io. https://akka.io Accessed on April 07, 2025.

[41] David H. Liu, Amit Levy, Shadi A. Noghabi, and Sebastian Burckhardt. 2023. Doing More with Less: Orchestrating Serverless Applications without an Orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 1505–1519. https://www.usenix.org/conference/nsdi23/presentation/liu-david

[42] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 11 (2020), 2047–2060. http://www.vldb.org/pvldb/vol13/p2047-lu.pdf

[43] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machine for WANs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 369–384. http://www.usenix.org/events/osdi08/tech/full_papers/mao/mao.pdf

[44] Microsoft. 2025. Azure Logic Apps. https://azure.microsoft.com/en-us/products/logic-apps Accessed on April 07, 2025.

[45] MinIO. 2025. MinIO. https://min.io Accessed on April 07, 2025.

[46] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 439–455. doi:10.1145/2517349.2522738

[47] Derek Gordon Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, David G. Andersen and Sylvia Ratnasamy (Eds.). USENIX Association. https://www.usenix.org/conference/nsdi11/ciel-universal-execution-engine-distributed-data-flow-computing

[48] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proc. VLDB Endow.* 10, 12, 1634–1645. doi:10.14778/3137765.3137770

[49] 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*.

[50] Kyriakos Psarakis, Wouter Zorgdrager, Marios Fragkoulis, Guido Salvaneschi, and Asterios Katsifodimos. 2024. Stateful Entities: Object-oriented Cloud Applications as Distributed Dataflows. In *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*, Letizia Tanca, Qiong Luo, Giuseppe Polese, Loredana Caruccio, Xavier Oriol, and Donatella Firmani (Eds.). OpenProceedings.org, 15–21. doi:10.48786/EDBT.2024.02

[51] Sheng Qi, Xuanzhe Liu, and Xin Jin. 2023. Halfmoon: Log-Optimal Fault-Tolerant Stateful Serverless Computing. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 314–330. doi:10.1145/3600006.3613154

[52] George Siachamis, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen, Paris Carbone, and Asterios Katsifodimos. 2024. CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 4030–4043. doi:10.1109/ICDE60146.2024.00309

[53] Pedro F. Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. 2021. Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1637–1650. doi:10.1145/3448016.3457320

[54] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael J. Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2021. DBOS: A DBMS-oriented Operating System. *Proc. VLDB Endow.* 15, 1 (2021), 21–30. doi:10.14778/3485450.3485454

[55] Jonas Spenger, Paris Carbone, and Philipp Haller. 2022. Portals: An Extension of Dataflow Streaming for Stateful Serverless. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2022, Auckland, New Zealand, December 8-10, 2022*, Christophe Scholliers and Jeremy Singer (Eds.). ACM, 153–171. doi:10.1145/3563835.3567664

[56] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 11 (2020), 2438–2452. http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf

[57] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. 2005. The 8 requirements of real-time stream processing. *SIGMOD Rec.* 34, 4 (2005), 42–47. doi:10.1145/1107499.1107504

[58] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27. http://sites.computer.org/debull/A13june/VoltDB1.pdf

[59] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 4–18. doi:10.1145/3514221.3526120

[60] Temporal. 2025. Introducing Temporal .NET – Deterministic Workflow Authoring in .NET. https://temporal.io/blog/introducing-temporal-dotnet. Accessed: 14-01-2025.

[61] 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 ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman (Eds.). ACM, 1–12. doi:10.1145/2213836.2213838

[62] Stephanie Wang, Eric Liang, Edward Oakes, Benjamin Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. 2021. Ownership: A Distributed Futures System for Fine-Grained Tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, James Mickens and Renata Teixeira (Eds.). USENIX Association, 671–686. https://www.usenix.org/conference/nsdi21/presentation/cheng

[63] 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 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 1187–1204. https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

[64] Shuhao Zhang, Juan Soto, and Volker Markl. 2024. A survey on transactional stream processing. *VLDB J.* 33, 2 (2024), 451–479. doi:10.1007/S00778-023-00814-Z

[65] Zihao Zhang, Huiqi Hu, Xuan Zhou, and Jiang Wang. 2022. STARRY: Multi-master Transaction Processing on Semi-leader Architecture. *Proc. VLDB Endow.* 16, 1 (2022), 77–89. doi:10.14778/3561261.3561268

[66] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2022. Lotus: Scalable Multi-Partition Transactions on Single-Threaded Partitioned Databases. *Proc. VLDB Endow.* 15, 11 (2022), 2939–2952. doi:10.14778/3551793.3551843

Received October 2024; revised January 2025; accepted February 2025