Styx in Action: Transactional Cloud Applications Made Easy

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

Oto Mraz
Delft University of Technology
o.m.mraz@tudelft.nl

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

George Siachamis
Inria & Institut Polytechnique de Paris
georgios.siachamis@inria.fr

Marios Fragkoulis
Delft University of Technology
m.fragkoulis@tudelft.nl

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

ABSTRACT

Developing and deploying transactional cloud applications such as banking and e-commerce systems is a daunting task for developers. The reason for this difficulty is twofold. First, developing such applications shifts the developers' focus from the application logic to considerations of distributed transactions, fault-tolerance, consistency, and scalability. Second, deploying such applications involves multiple systems, such as databases, load balancers, or containerized services, impeding efficient resource management.

This demonstration presents Styx, a scalable application runtime that allows developers to build scalable and transactional cloud applications with minimal effort. It supports serializability and exactly-once guarantees and focuses on the ease of development and deployment, as well as Styx's fault-tolerance mechanisms.

PVLDB Reference Format:
Kyriakos Psarakis, Oto Mraz, George Christodoulou, George Siachamis, Marios Fragkoulis, and Asterios Katsifodimos. Styx in Action: Transactional Cloud Applications Made Easy. PVLDB, 18(12): 5275 - 5278, 2025.
doi:10.14778/3750601.3750650

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/delftdata/styx.

1 INTRODUCTION

The Function-as-a-Service (FaaS) paradigm has gained significant popularity as it simplifies the use of cloud resources with minimal programmer effort. FaaS offers a serverless service model without operational burden and a pay-as-you-go pricing model. FaaS is based on the storage and execution disaggregation design principle, where a stateless function layer executes application logic while an external store handles the state, ensuring consistency and persistence. However, this separation introduces challenges for applications that require strong transactional guarantees, including banking systems and e-commerce platforms. Such systems require coordination and communication between the stateless execution layer and the stateful database to ensure data integrity and reliable execution. Unfortunately, this architecture introduces overhead and error handling, which pollutes the application logic.

These challenges can be exemplified with an online shopping cart application. Multiple steps must be executed in sequence when completing a checkout: verifying product availability, processing a payment, and shipping the corresponding order. In a microservice-like architecture, each service (e.g., Cart, Stock, and Payment) operates independently with its own API, database, and business logic, communicating via API calls. Ensuring atomicity, such that stock updates and payments succeed or fail together, and maintaining state consistency across multiple workflows, ensures that stock counts accurately reflect successful payments. This results in complex application logic. Handling such complexities at the application level can be error-prone and inefficient, imposing programming and operational overheads on developers.

A Stateful FaaS (SFaaS) service model can address the key challenges of cloud applications by eliminating the need for developers to handle application state manually, ensuring resilience and seamless recovery from failures. Moreover, supporting transactionality in SFaaS can guarantee end-to-end state consistency across multiple functions. To this end, we argue that a suitable SFaaS runtime for transactional applications should be built on the following three principles: *i*) a high-level programming model, allowing developers to avoid dealing with low-level primitives like locks, transaction coordination and failure handling; *ii*) high performance with low-latency and high-throughput execution; *iii*) ensure exactly-once state mutations, with serializable guarantees, even in the presence of failures or retries.

Recently, we introduced Styx [5], a dataflow-based runtime designed for transactional cloud applications built on the aforementioned principles. Styx ensures that each transaction's state mutations are reflected in the system's state exactly-once, even under failures, retries, or other potential disruptions. Additionally, it supports arbitrary function orchestrations with end-to-end serializability by leveraging a deterministic database protocol, eliminating the need for expensive two-phase commits. Our approach is inspired by two key observations [4]. First, modern streaming dataflow systems such as Apache Flink [1] guarantee exactly-once processing by transparently handling failures. However, these systems cannot execute general cloud applications and do not support transactional function orchestrations. Second, efficient transaction execution on top of dataflow systems can be enabled through deterministic database protocols, such as Calvin [8] or Aria [3], without the overhead of two-phase commits. Styx bridges this gap by integrating a deterministic transactional protocol that allows early commit replies to clients, improving responsiveness while maintaining consistency.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750650

Demonstration Scenarios. To illustrate the capabilities of Styx, in our demonstration we focus on three scenarios. **(1)** We demonstrate the developer experience by showing how application logic can be free of transaction management and failure handling code. To this end, we have integrated a compiler for transforming object-oriented programs into dataflows optimized for our runtime [6]. **(2)** We highlight the system's deployment and rescaling capabilities, demonstrating how these processes can be performed with minimal overhead. **(3)** We showcase how Styx seamlessly recovers from worker failures without affecting application performance. Additionally, the Styx UI provides live system metrics, offering attendees real-time visibility into system operations.

2 THE STYX RUNTIME

Styx is a transactional distributed dataflow system that executes workflows of stateful functions with serializable guarantees. Styx's higher-level programming abstraction, Stateflow [6], enables users to code in a pure object-oriented style without state management or fault tolerance logic. In this section, we describe the programming model (Section 2.1), runtime design (Section 2.2), and fault tolerance mechanism (Section 2.3).

2.1 Programming Model

The dataflow model decomposes programs into independent processing units, organized as nodes in Directed Acyclic Graphs (DAGs) that exchange data through message streams (edges). Dataflows have been used as a programming model for analytical batch and stream processing systems like Flink, where processing units are denoted as operators performing either stateful (e.g., joins, aggregates) or stateless (e.g., map, filter) operations. Nevertheless, integrating the dataflow model with transactional cloud applications is challenging. Dataflow systems typically require developers to rewrite cloud applications to fit the event-driven dataflow model.

Styx provides developers with two levels of abstraction: a high-level actor-like programming interface based on Stateflow [6] and a lower-level dataflow API [5].

High-level Programming Model. Users can code transactional cloud applications in Python object-oriented code. Entities are objects that contain unique key and class functions that mutate state. Additionally, when a function call to another entity occurs, Stateflow automatically creates an edge in the dataflow graph. The transformation of continuation-passing style programming to entity calls in a distributed dataflow graph is described in [6].

Low-level Programming Model. Styx follows the operator API of dataflow systems like Apache Flink [1]. In Styx, a streaming operator can hold multiple entities based on a partitioning scheme, the functions that act upon the operator as a whole (allowing for range queries), or the entities themselves (point queries). To communicate across operators, developers can call remote operator functions using Styx's API.

2.2 Runtime Design Overview

Styx [5] (Figure 1) employs a worker/coordinator architecture combined with a messaging system, such as Apache Kafka, that propagates input into Styx while being responsible for replaying uncommitted messages after a failure. The coordinator's responsibilities

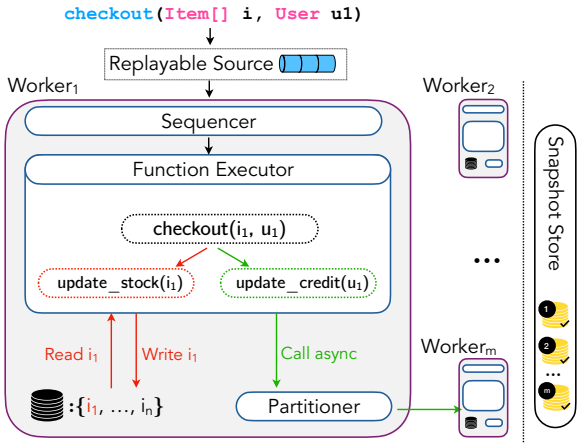


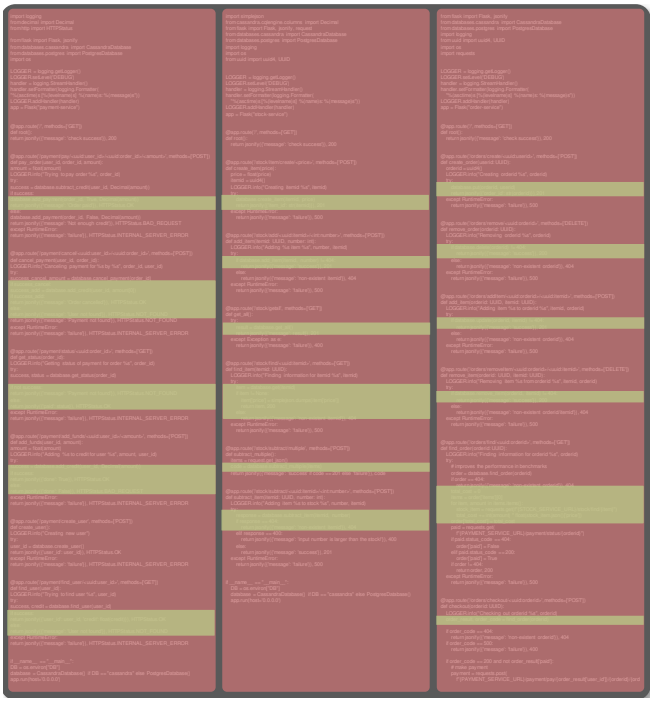
Figure 1: Stateful-Function execution in Styx.

are to deploy a user-defined dataflow graph to the workers and trigger the fault tolerance pipeline in case of failure. Furthermore, it monitors the cluster's health and collects metrics, as shown in the current demonstration.

Each worker is responsible for a subset of the dataflow graph's operator state partitions, which are 1-to-1 aligned with the partitions of the replayable input source. Client requests are ingested and sequenced via a partitioned sequencer per worker. Afterward, batches of transactions are executed as coroutines in a single CPU per worker to increase efficiency. Styx employs an epoch-based deterministic transactional protocol to execute transactions in a deterministic fashion. Each epoch is divided into four stages. In the first stage, IDs (TIDs) are assigned to transactions. In the second stage, transactions are executed to obtain each transaction's read-write set. In the third phase, transactions that do not participate in unresolved conflicts caused by their state accesses are committed. In the final phase, transactions that participate in conflicts are executed with a lock-based mechanism. Transactions are executed in parallel across workers, and nested function calls are transparently scheduled for execution by local or remote operators. Since the programming model of Styx allows for arbitrary function execution, its acknowledgment-sharing mechanism is required to identify the transactions' boundaries. Finally, Styx enables optimizations through determinism, guaranteeing the same state mutations after replay in the event of a failure. For instance, Styx gains the ability to reply to a client even before a persistent snapshot is stored.

2.3 Fault Tolerance

Fault tolerance is a concern for both transactional applications and dataflow systems. Existing SFaaS systems achieve fine-grained fault tolerance by leveraging logging. Current approaches log and persist each function call, introducing significant overhead during execution. Dataflow systems enable coarse-grained fault tolerance through checkpointing protocols [7], possibly combined with logging mechanisms. These systems recover by restoring their state from the latest valid checkpoint and replaying missed messages



(a) Microservice implementation using the saga pattern (Red: code to ensure atomicity and fault tolerance, Green: business logic).

```python
from styx import Operator, StatefulFunction
from shopping_cart.operators import stock, payment, cart
from shopping_cart.exceptions import NotEnoughCredit, NotEnoughStock

@stock.register
async def decrement_stock(ctx: StatefulFunction, amount: int):
    item_stock = ctx.get()
    item_stock -= amount
    if item_stock < 0:
        raise NotEnoughStock(f"Item: {ctx.key} does not have enough stock")
    ctx.put(item_stock)

@payment.register
def pay(ctx: StatefulFunction, amount: int):
    credit = ctx.get()
    credit -= amount
    if credit < 0:
        raise NotEnoughCredit(f"User: {ctx.key} does not have enough credit")
    ctx.put(credit)

@cart.register
def checkout(ctx: StatefulFunction):
    items, user_id, total_price, paid = ctx.get()
    for item_id, qty in items:
        ctx.call_async(operator=stock,
                       function_name='decrement_stock',
                       key=item_id,
                       params=(qty, ))
    ctx.call_async(operator=payment,
                   function_name='pay',
                   key=user_id,
                   params=(total_price, ))
    paid = True
    ctx.put((items, user_id, total_price, paid))
    return "Checkout Successful"
```

(b) Checkout workflow in Styx.

Figure 2: Comparison between the microservice paradigm (fig. 2a) and Styx (fig. 2b).

from the logs. With coordinated checkpointing [2], additional logging may not be needed, as processing can resume from the correct offset.

Styx relies on the input source and determinism for message replay based on recorded offsets. This design ensures that the sequencer will recreate the same transaction sequence post-recovery, enabling early replies (before the state commits to durable storage) and exactly-once guarantees. Styx utilizes a blob store for durable storage to persist incremental snapshots of worker states.

3 DEMONSTRATION OVERVIEW

We now describe how we showcase the key capabilities of Styx, as discussed in Section 2, and how attendees can engage with it. To this end, we present three scenarios: one demonstrating the benefits for developers during application development and two highlighting the performance advantages of Styx in practice. To visualize these performance benefits, we have designed and implemented a dashboard that monitors Styx.

3.1 Scenario 1: Application Development

Figure 2 showcases the difference in developing a simplified shopping cart application with three services (stock, payment, cart) between a traditional microservice implementation and Styx. Styx eliminates the boilerplate code needed to ensure ACID guarantees in the microservice implementation and allows developers to focus solely on the core application logic. Beyond accelerating development, Styx enhances maintainability and reduces the likelihood of bugs by providing serializable transactions as a service. This results in cleaner, more reliable code, ultimately achieving long-term software quality. Attendees can change parts of the application code and submit applications to the Styx runtime.

3.2 Scenario 2: Deployment and Rescaling

Styx is designed for seamless deployment and rescaling. To facilitate real-time monitoring, we have developed two dashboards. In the depicted scenario, we execute the YCSB-T workload across four Styx workers at 10.000 transactions per second (TPS) following a uniform distribution within one million keys.

Part 1: System Overview. Attendees will be able to assess system performance across all Styx workers. The *System Overview* dashboard Figure 3 provides a high-level summary of key system metrics:

- **Resource Metrics (A)**: Displays the average CPU and memory utilization, as well as ingress and egress network traffic across Styx workers.
- **Performance Metrics (B)**: Visualizes transaction throughput per second, average transaction latency, and abort rate. Under normal conditions, epoch latency (Styx uses a deterministic epoch-based commit protocol) for the YCSB-T workload remains below 250 ms (green-shaded region). At the same time, the abort rate fluctuates based on the level of contention from 0% (no contention) to 100% (all transactions within an epoch contain the same key at least once).

Figure 3: Styx monitoring dashboards.

- **Latency Breakdown (C)**: A pie chart categorizes transaction latency into distinct components. Typically, the primary contributors to latency are the first optimistic transaction execution (1st Run) with the call-graph discovery (Chain Acks), the lock-based fallback commit mechanism (Fallback), and others like cross-worker synchronization, write-ahead-logging (WAL), conflict resolution + commit, and the asynchronous snapshots.
- **Snapshot Latency (D)**: Time taken for a complete delta snapshot throughout the deployment.
- **Worker Health (E)**: The final panel tracks time since the last heartbeat. If this value remains below 1000 ms (green-shaded), it confirms that all workers are healthy and operational.
- **Reconfiguration (G1-3)**: At "19:19:30", we downscale the deployment from four partitions to three, and we observe an increase in snapshot latency and ingress network (data transferred across workers through S3), and finally a decrease in TPS since we decreased the parallelism.

The demonstration attendees will be able to change the skew factors of the supported workloads and perform updates to observe changes in the performance (latency, throughput) of Styx applications in real time.

Part 2: Worker Specific. With a global view of the system's health in mind, attendees can drill down into the performance of individual workers using the *Worker Specific* dashboard. This dashboard mirrors the system overview but focuses on a selected Styx worker. A **drop-down menu (F)** allows attendees to choose a specific worker, enabling direct comparison with the overall system metrics. By analyzing the worker-specific metrics, attendees can quickly pinpoint anomalies. If a single worker exhibits significantly higher transactional latency or reduced throughput compared to the others, it may indicate an overloaded or unhealthy state.

3.3 Scenario 3: Fault Tolerance

The final scenario showcases Styx's ability to handle failures efficiently. During the demonstration, attendees can manually terminate a Styx worker to observe how the system detects the failure and triggers its recovery process.

Part 1: System Overview. The system overview dashboard provides a real-time visual indicator of worker failures. When a Styx worker stops responding, the *Time Since Last Heartbeat* metric (panel E) spikes, signaling the loss of communication. This event is accompanied by a sharp increase in transactional latency and a temporary dip in throughput until the system fully recovers. Once the operators assigned to the dead worker are rescheduled to a new or existing worker, Styx begins handling the delayed transactions, and these metrics gradually return to their normal ranges.

Part 2: Worker Specific. Using the worker specific dashboard, attendees can further investigate the failed worker's behavior. The impact of the failure is more pronounced here. Transaction *latency* and throughput fluctuations become more drastic, and for a brief period, the failed worker will stop reporting metrics entirely. Once the recovery process is completed, these values stabilize, confirming that the system has successfully recovered. This scenario demonstrates Styx's resilience and self-healing mechanisms, ensuring system reliability even in the event of failures.

ACKNOWLEDGMENTS

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

REFERENCES

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

[2] K Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[3] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 12 (July 2020), 2047–2060.

[4] 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.*

[5] 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 (2025).

[6] Kyriakos Psarakis, Wouter Zorgdrager, Marios Fragkoulis, Guido Salvaneschi, and Asterios Katsifodimos. 2024. Stateful Entities: Object-oriented Cloud Applications as Distributed Dataflows. In *Proc. 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy.* 15–21.

[7] George Siachamis, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen, Paris Carbone, and Asterios Katsifodimos. 2024. CheckMate: Evaluating Checkpointing Protocols for Streaming Dataflows. In *ICDE 2024.* 4030–4043.

[8] 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 *SIGMOD 2012 (SIGMOD '12).* Association for Computing Machinery, New York, NY, USA, 1–12.